Low Level Exploits

Email: hughpearse@gmail.com
Twitter: https://twitter.com/hughpearse
LinkedIn: http://ie.linkedin.com/in/hughpearse

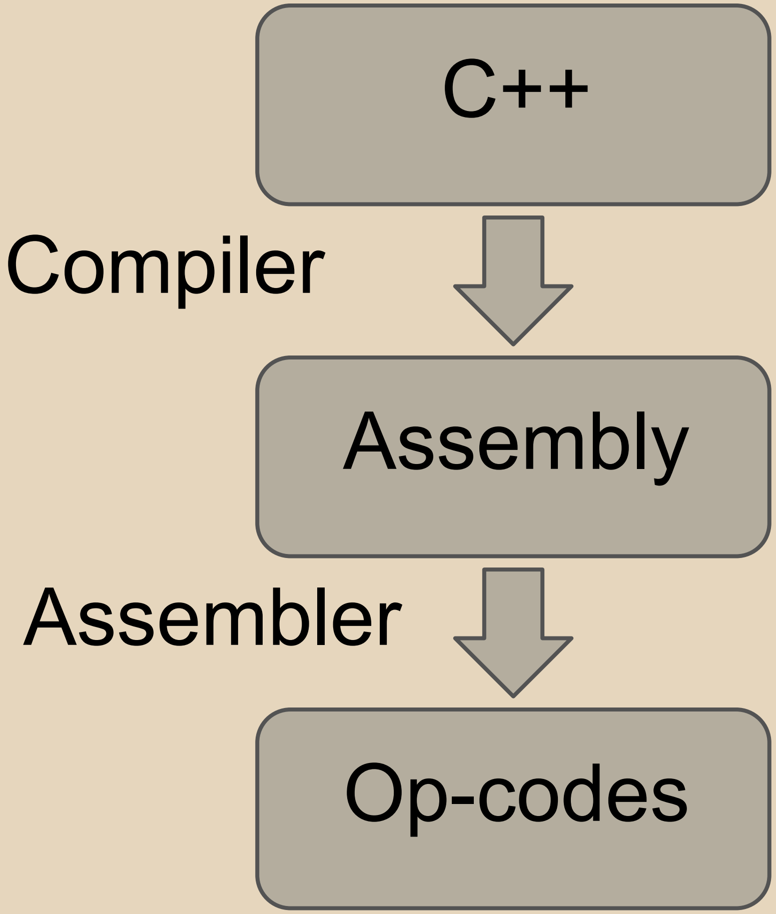
Table of Contents

Creating Shellcode

Creating Shellcode

Natively Compiled Code does not normally run in an interpreter such as a JVM.

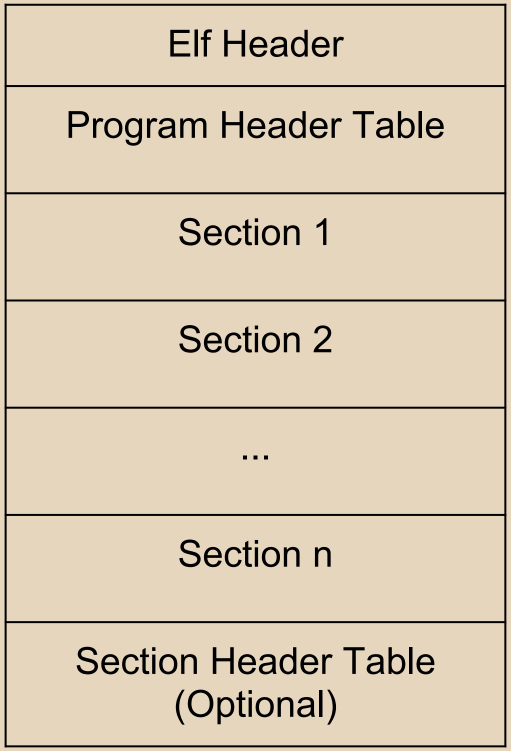
```
#include<stdio.h>
main(){
    printf("hello, world");
}
```



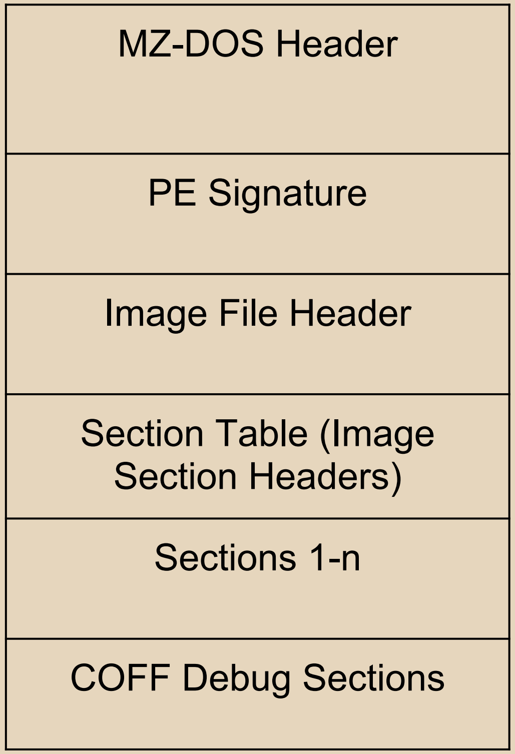
Creating Shellcode

The ELF file typically runs on a Linux/Apple/UNIX while the PE file typically runs on Windows.

Elf File Format



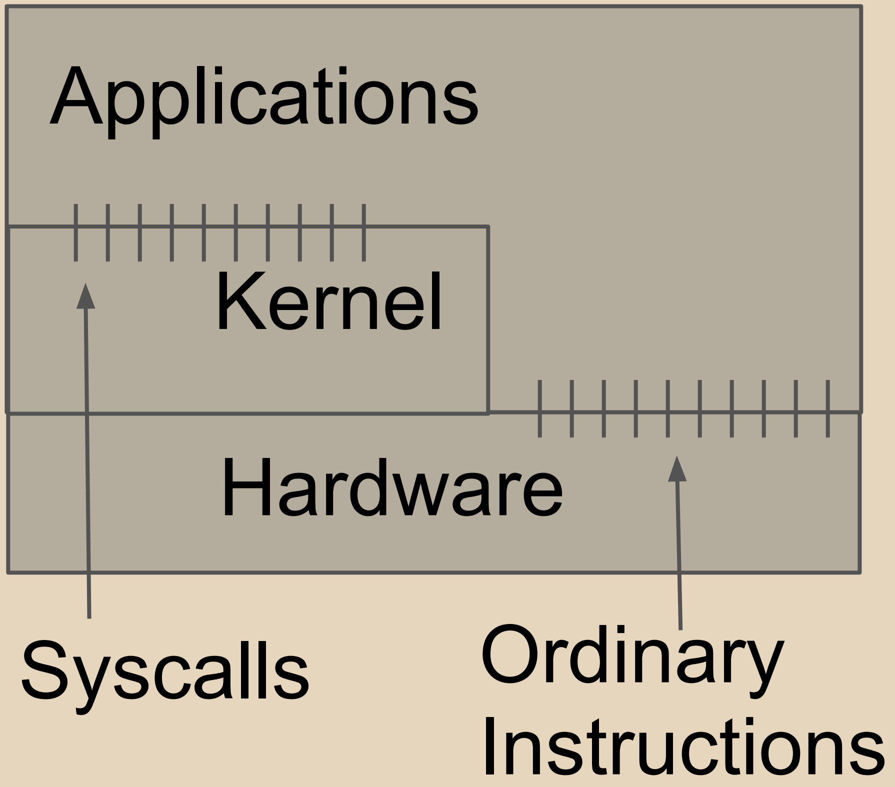
PE File Format



Creating Shellcode

When a linux application executes the integer 0x80 it causes a software interrupt.

The kernel then takes over by taking the values in the general registers in the CPU and launching the appropriate system call based on the values in the registers.



Creating Shellcode

Let us write an application to execute the exit syscall.

```
//exit.c
main(){
    exit(0);
}
```

Compile the program using the following command:

```
gcc -static exit.c -o exit.out
```

This command creates a file called "exit.out".
Lets look at the contents of this file.

Creating Shellcode

objdump -d ./exit

```
08048f14 <main>:
8048f14:        55                      push   %ebp
 8048f15:       89 e5                   mov    %esp,%ebp
 8048f17:       83 e4 f0                and    $0xfffffff0,%esp
 8048f1a:       83 ec 10                sub    $0x10,%esp
 8048f1d:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048f24:       e8 77 08 00 00          call   80497a0 <exit>
 8048f29:       66 90                   xchg   %ax,%ax
 8048f2b:       66 90                   xchg   %ax,%ax
 8048f2d:       66 90                   xchg   %ax,%ax
 8048f2f:       90                      nop

08053a0c <_exit>:
8053a0c:    8b 5c 24 04                 mov    0x4(%esp),%ebx
 8053a10:   b8 fc 00 00 00              mov    $0xfc,%eax
 8053a15:   ff 15 a4 f5 0e 08           call   *0x80ef5a4
 8053a1b:   b8 01 00 00 00              mov    $0x1,%eax
 8053a20:   cd 80                       int    $0x80
 8053a22:   f4                          hlt
 8053a23:   90                          nop
 8053a24:   66 90                       xchg   %ax,%ax
 8053a26:   66 90                       xchg   %ax,%ax
 8053a28:   66 90                       xchg   %ax,%ax
 8053a2a:   66 90                       xchg   %ax,%ax
 8053a2c:   66 90                       xchg   %ax,%ax
 8053a2e:   66 90                       xchg   %ax,%ax
```

Creating Shellcode

The output of the "objdump" command has three columns. Virtual addresses, Op-codes and Mnemonics.

We want the opcodes to create our payload.

We will be storing our shellcode inside a character array. This means we cannot have null values.

Also sometimes endianness can be a problem.

Creating Shellcode

Virtual addresses, Op-codes, Mnemonics

```
08048f14 <main>:

8048f14:         55                    push   %ebp
 8048f15:        89 e5                 mov      %esp,%ebp
 8048f17:        83 e4 f0                       and       $0xfffffff0,%esp
 8048f1a:        83 ec 10                       sub       $0x10,%esp
 8048f1d:        c7 04 24 00 00 00 00           movl   $0x0,(%esp)
 8048f24:        e8 77 08 00 00                 call   80497a0 <exit>
 8048f29:        66 90                 xchg   %ax,%ax
 8048f2b:        66 90                 xchg   %ax,%ax
 8048f2d:        66 90                 xchg   %ax,%ax
 8048f2f:        90                    nop


08053a0c <_exit>:

8053a0c:     8b 5c 24 04                         mov     0x4(%esp),%ebx
 8053a10:    b8 fc 00 00 00                      mov     $0xfc,%eax
 8053a15:    ff 15 a4 f5 0e 08        call   *0x80ef5a4
 8053a1b:    b8 01 00 00 00                      mov     $0x1,%eax
 8053a20:    cd 80                               int       $0x80
 8053a22:    f4                       hlt
 8053a23:    90                                  nop
 8053a24:    66 90                               xchg   %ax,%ax
 8053a26:    66 90                               xchg   %ax,%ax
 8053a28:    66 90                               xchg   %ax,%ax
 8053a2a:    66 90                               xchg   %ax,%ax
 8053a2c:    66 90                               xchg   %ax,%ax
 8053a2e:    66 90                               xchg   %ax,%ax
```

Creating Shellcode

Our op-codes should look like this:
8b 5c 24 04 b8 fc 00 00 00 ff 15 a4 f5 0e 08 b8 01 00 00 00 cd 80

Paste this text into a text editor and use the "find and replace" feature to replace space with "\x" to make a hexadecimal character array out of it. Don't forget to surround the text with quotes.

```
"\x8b\x5c\x24\x04\xb8\xfc\x00\x00\x00\xff\x15\xa4\xf5\x0e\x08\xb8\x01\x00\x00\x00\xcd\x80"
```

We now have a string containing machine instructions.
Notice the \x80 at the end of the string?

Creating Shellcode

Simply execute the character array to test it.

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

char* shellcode = "\x8b\x5c\x24....."
int main(){
    void (*f)();
    f = (void (*)())shellcode;
    (void)(*f)();
}
```

Buffer Overflows On the Stack

Buffer Overflows On the Stack

When an application is launched three import program sections are created in memory. These sections contain different types of information.

1. Text -> Stores machine instructions
2. Stack -> Stores automatic variables and return addresses
3. Heap -> Stores variables whose size is only known at runtime by using the malloc() function.

For the moment we are interested in the Stack.

Buffer Overflows On the Stack

Stack Frames

The stack is divided up into contiguous pieces called frames. A frame is created each time a function is called.

A frame contains:

1. the arguments to the function.
2. the function's local variables.
3. the address at which the function is executing.
4. the address at which to return to after executing.

Buffer Overflows On the Stack

When the program starts, the stack has only one frame for main(). This is called the initial frame or the outermost frame.

Each time a function is called, a new frame is made.

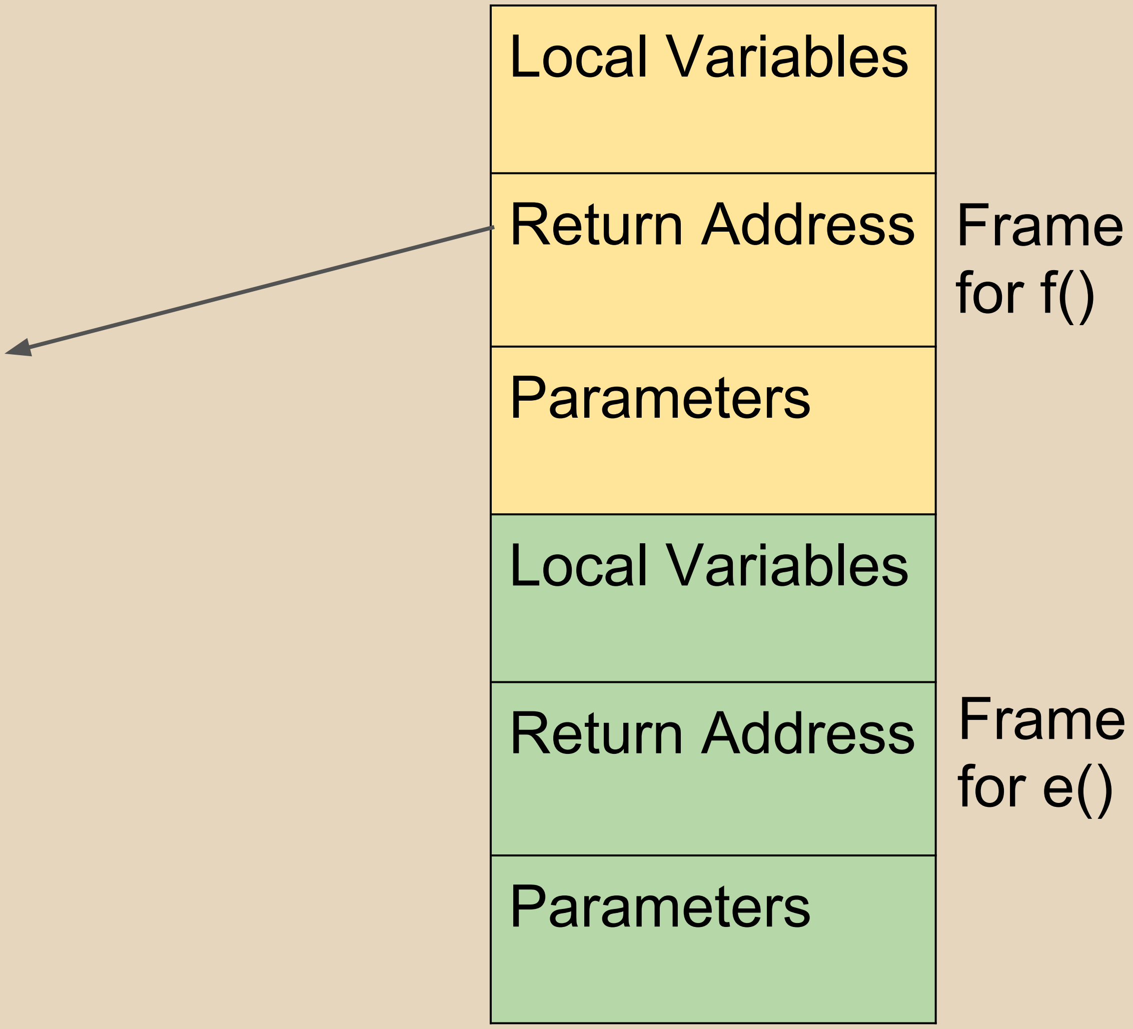
Each time a function returns, the frame for that function is eliminated.

If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the innermost frame. This is the most recently created of all the stack frames that still exist.

Buffer Overflows On the Stack

```
//example-functions.h
void e(int a1, int a2){
	int local1=1;
	int local2=2;
	f(local1, local2);
}

void f(int a1, int a2){
	int local1;
	int local2;
}
```



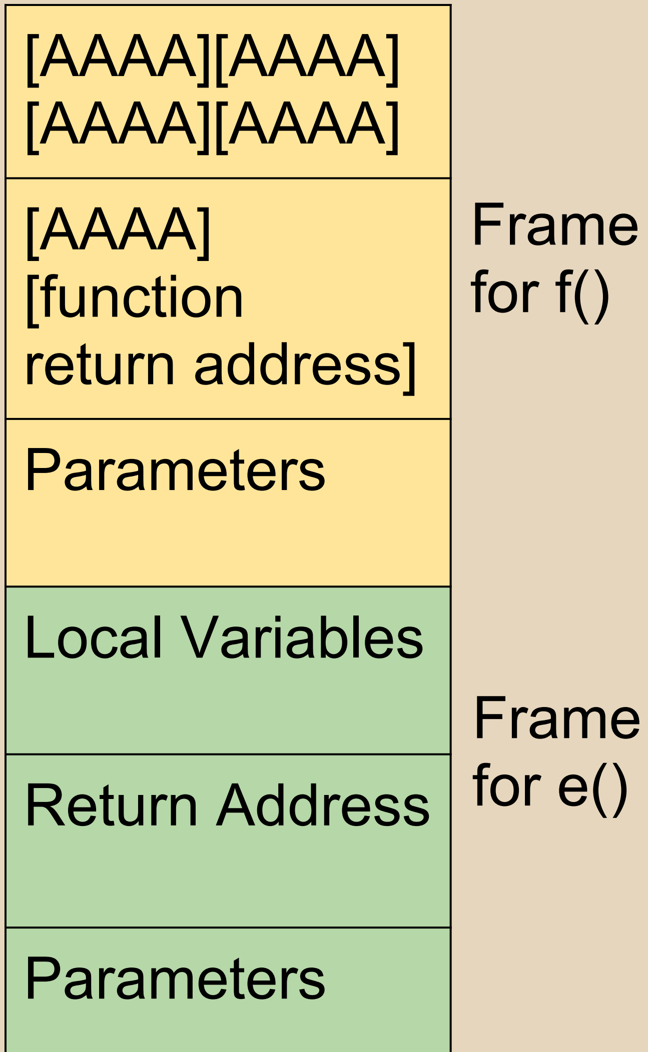
Buffer Overflows On the Stack

Find the address of other functions that were statically compiled into the program by using the command:

```
gdb -q -ex "info functions" --batch
./hello-world | tr -s " " |
cut -d " " -f 2 | sort | uniq
```

[AAAA][AAAA][AAAA][AAAA]
[AAAA][function return address]

Function F does not return to E;
Function F returns to X.



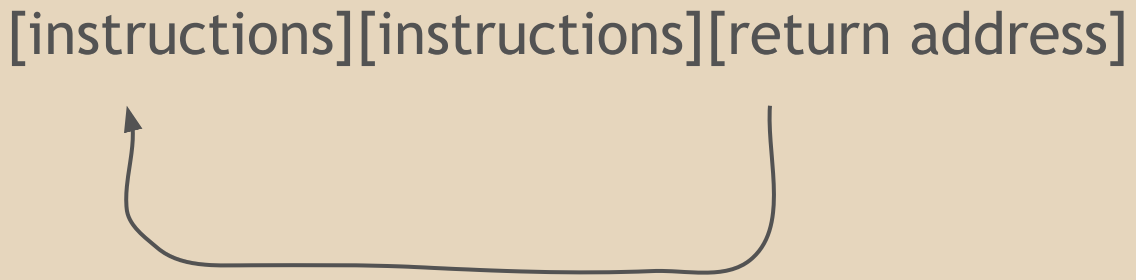
Return to Stack

Return to Stack

Return to Stack

A buffer overflow attack is a general definition for a class of attacks to put more data in a buffer than it can hold thus overwriting the return address.

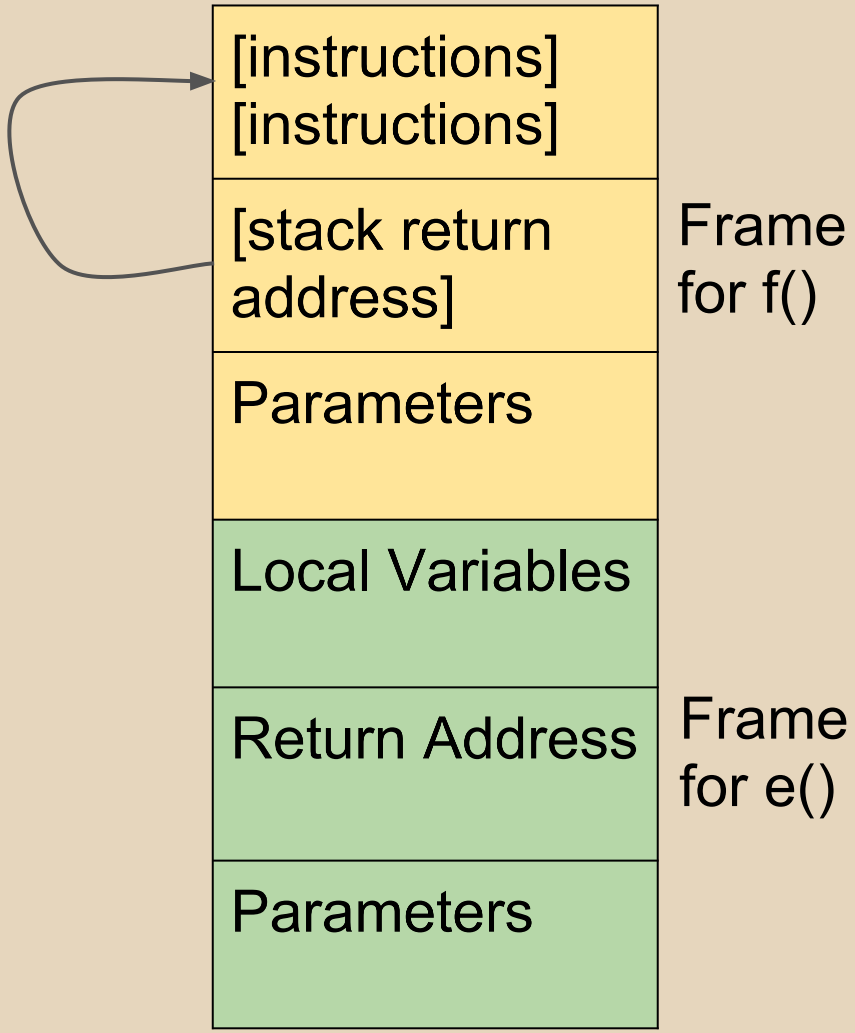
Return to Stack is a specific stack buffer overflow attack where the return address is the same as the address on the stack for storing the variable information. Thus you are executing the stack.



Return to Stack

As you can see in the diagram the local variables of the function have been overwritten.

The Return to Stack exploit is also known as "Smashing The Stack".



Format String Vulnerabilities

Format String Vulnerabilities

What is a format string?

```
int x=3;
printf("%d", x);
```

How many formats exist in C/C++?

There are at least 17 basic ones, and lots of permutations.

- %d Signed decimal integer
- %s Character string
- %x Unsigned hexadecimal integer
- %u Unsigned decimal integer
- %n The number of characters written so far

Format String Vulnerabilities

How do format string vulnerabilities occur?
They are usually lazy programmers who make mistakes.

This is correct:

```
    printf("%s", argv[1]);
```

This is incorrect:

```
    printf(argv[1]);
```

```
#include <stdio.h>
void main(int argc, char *argv[]){
    printf("%s", argv[1]);
}
```

Format String Vulnerabilities

But they both seem to work?
Yes! It will work which makes it difficult to detect.

```
./correct "Hello123"
Output: Hello123
```

```
./incorrect "Hello123"
Output: Hello123
```

Format String Vulnerabilities

Lets try inputting format string specifiers

```
./correct "%x"
Output: %x

./incorrect "%x"
Output: 12ffb8
```

Format String Vulnerabilities

Lets take another look at what format string specifiers can do.

%x - Pop data off the stack and display it

%s - Dereference pointer seen above and read to null byte value

%n - Dereference counter location and write the functions output count to address

%x - Read from stack

%s - Read from memory

%n - Write to memory

Format String Vulnerabilities

Exploiting the vulnerability

```
./incorrect "AAAA %x %x %x %x %x"
Output: AAAA 12ffb89a 12f376 77648426 41414141
```

Return to LibC

Return to LibC (Linux)

"Smash the stack" attacks have been made more difficult by a technology called a non-executable stack.

NX stack can still be bypassed.

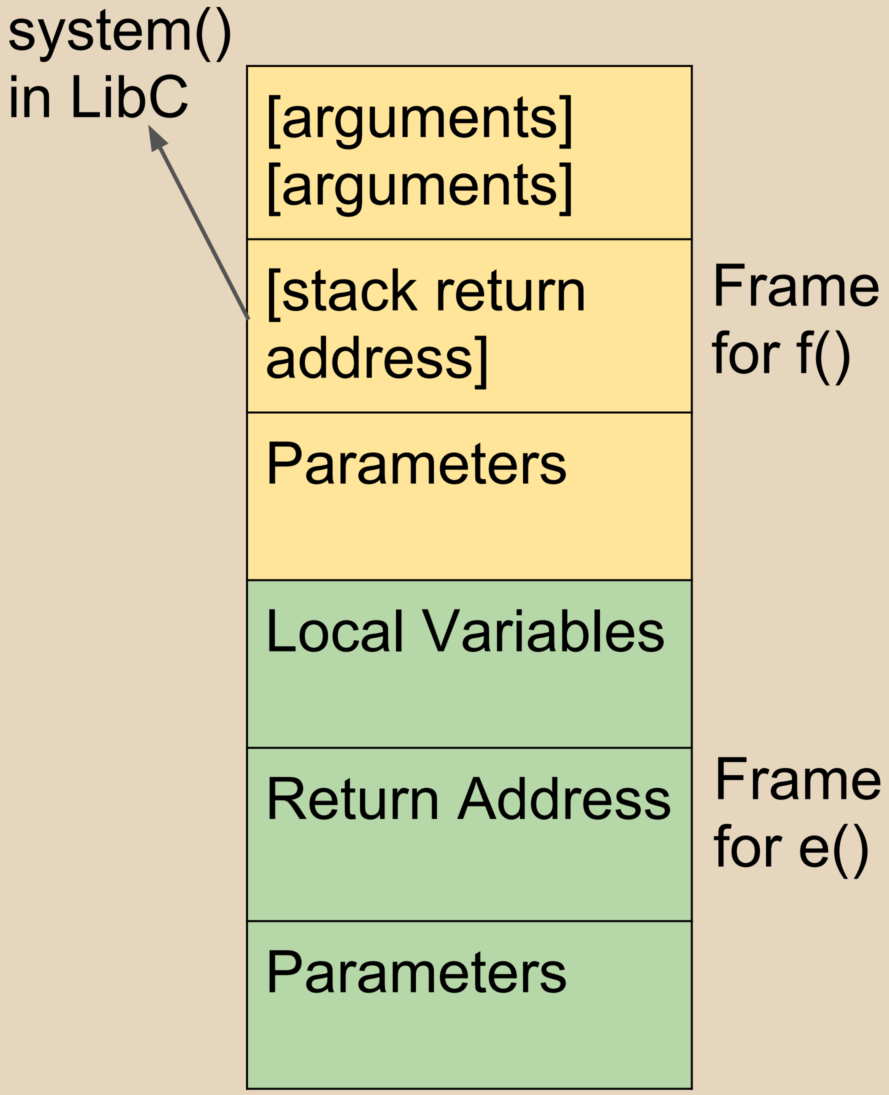
A buffer overflow is still required but the data on the stack is not executable shellcode, it contains read-only arguments to a function.

LibC is a dynamic library that is part of every userspace application on a linux system. This library contains all sorts of functions such as system() to launch an application.

Return to LibC (Linux)

Now we can spawn a shell

system("/bin/bash wget ...");



Return to Structured Exception Handler (SEH)

Return to SEH (Windows)

Stack cookies are numbers that are placed before the return address when a function begins, and checked before the function returns. This prevents buffer overflow attacks.

[buffer][cookie][saved EBP][saved EIP]

Return to SEH (Windows)

If the overwritten cookie does not match with the original cookie, the code checks to see if there is a developer defined exception handler. If not, the OS exception handler will kick in.

[buffer][cookie][SEH record][saved ebp][saved eip]

(1.) - Overwrite an Exception Handler registration structure
(2.) - Trigger an exception before the cookie is checked
(3.) - Return to overwritten Exception Handler

Return to Heap

Return to Heap (Heap Spraying)

Heap spraying is an unreliable method of increasing the chances of returning to executable instructions in a buffer overflow attack.

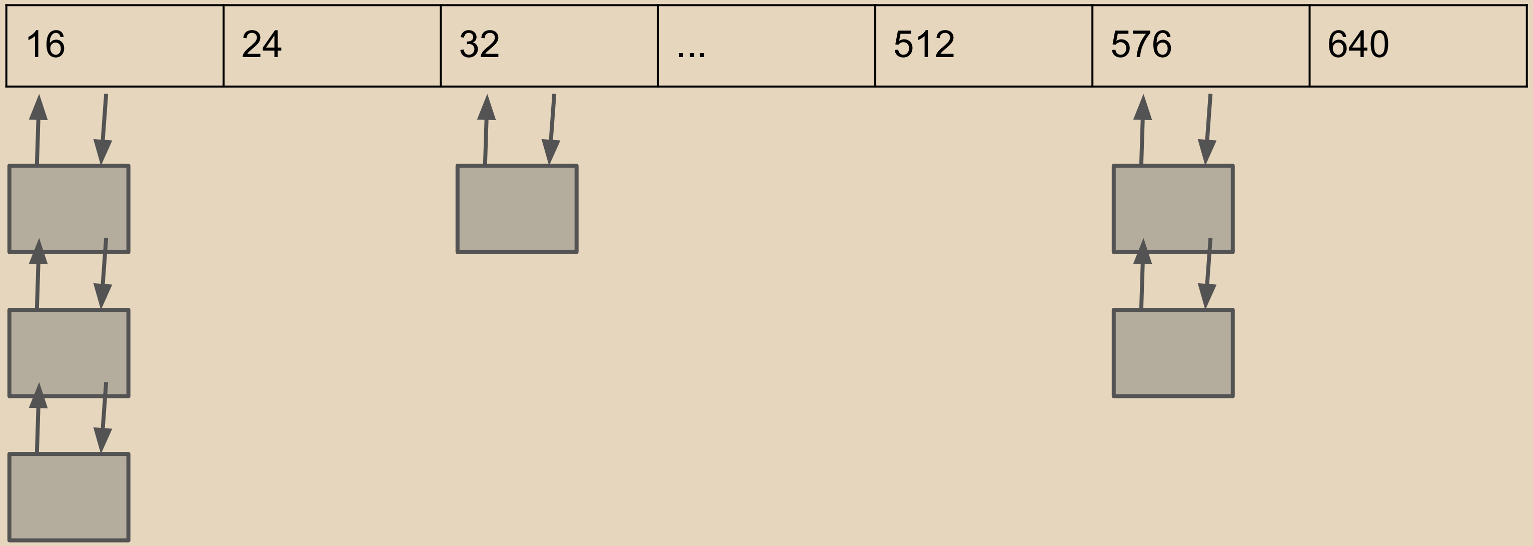
Attackers create thousands of data structures in memory which contain mostly null operations (the 0x90 instruction) with the executable machine instructions at the end of the structure.

Statistically the chances of returning to a valid location on a "NOP sled" are increased by increasing the size of the data structures. Sometimes the chance can be up to 50%.

Heap Overflows

unlink() Technique by w00w00

The Bin structure in the .bss segment stores unused memory chunks in a doubly linked list. The chunks contain forward pointers and backward pointers.



unlink() Technique by w00w00

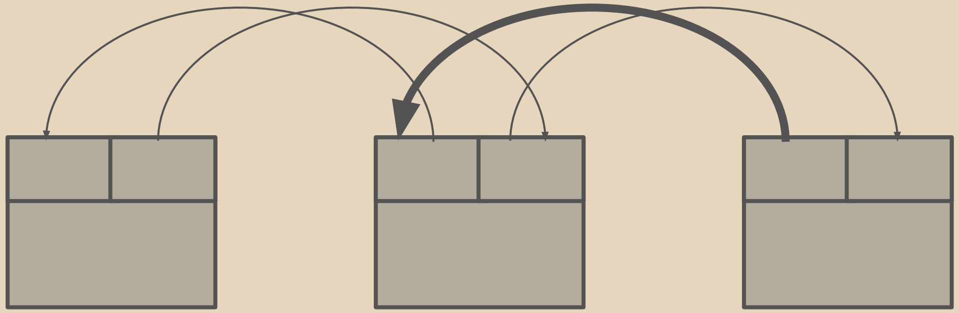
When memory is allocated, the pointers of the previous and next block are re-written using the unlink() function by de-referencing the pointers in the middle block.

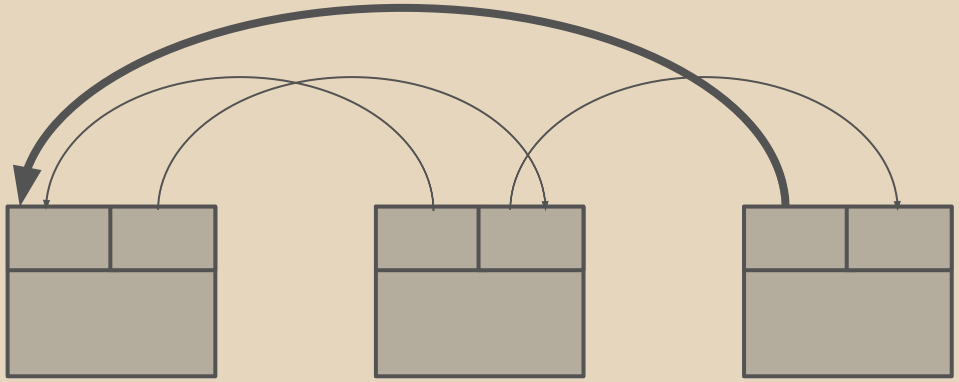
```
#define unlink( P, BK, FD ) {
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
}
```

unlink() Technique by w00w00

How a normal unlink() works.

Step 1:

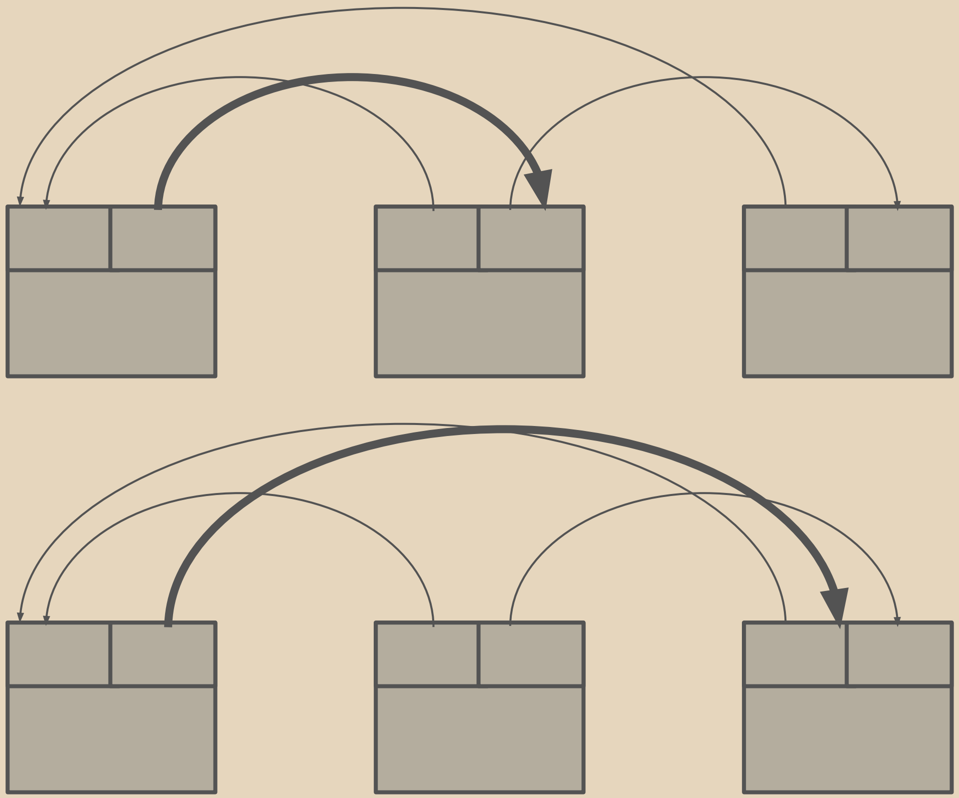




unlink() Technique by w00w00

How a normal unlink() works.

Step 2:



unlink() Technique by w00w00

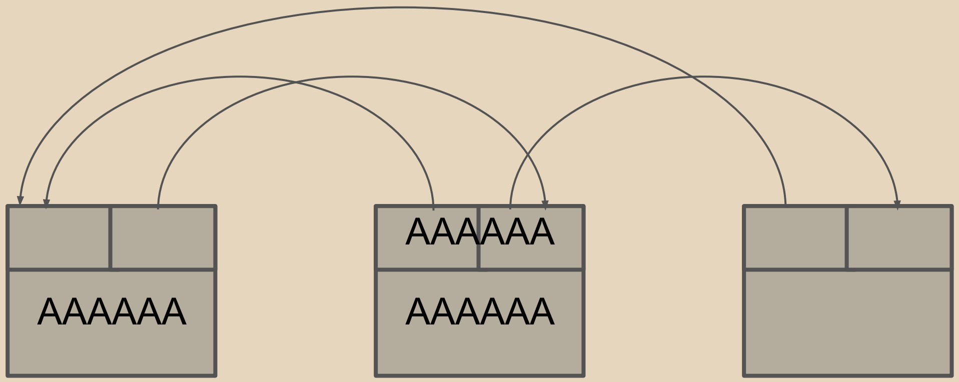
When a heap management operation (such as free / malloc / unlink) is made, pointers in the chunk headers are dereferenced.

We want to target the unlink() macro (a deprecated version of the Doug Lea memory allocator algorithm).

unlink() Technique by w00w00

Heap overflows work by overflowing areas of heap memory and overwriting the headers of the next area in memory.

If the first chunk contains a buffer, then we can overwrite the headers in the next chunk which is unallocated.



unlink() Technique by w00w00

By altering these pointers we can write to arbitrary addresses in memory. Dereferencing is confusing!

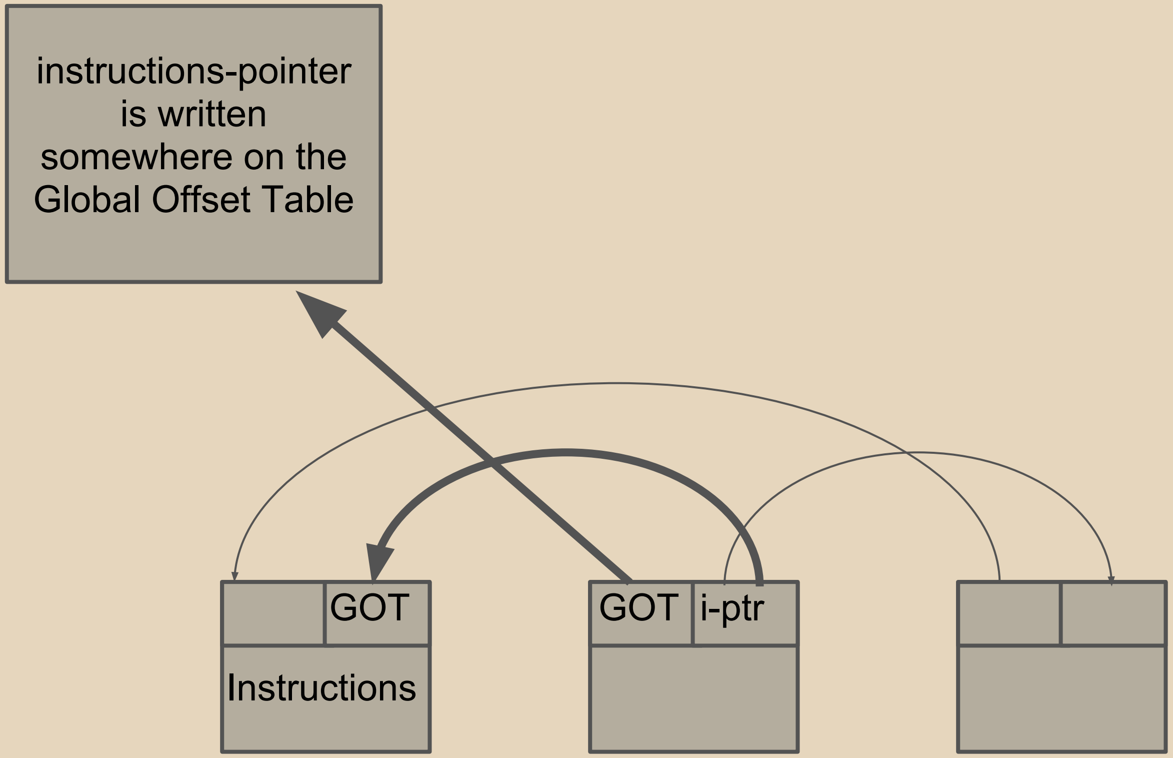
Unlink has 2 write operations. By overwriting the header of a chunk, we can choose what we write to memory!

[fwd-ptr] = bk-ptr
    and
[bk-ptr] = fwd-ptr

Try overwriting pointers in the Global Offset Table.

unlink() Technique by w00w00

We can force unlink() to write to the GOT, and write to the shellcode chunk.


instructions-pointer is written somewhere on the Global Offset Table
GOT
Instructions
GOT
i-ptr

unlink() Technique by w00w00

The memory pointed to by fd+12 is overwritten with bk, then the memory pointed to by bk+8 is overwritten with the value of fd.

unlink() overwrites the GOT with the shellcode's address in the first step. This was the primary goal of the exploit.

The second step writes a pointer just past the start of the shellcode. This would normally render the shellcode unrunnable, but the shellcode can be made to start with a jump instruction, skipping over the part of the shellcode that is overwritten during unlinking.

Integer Overflows

Integer Overflows

Primitive Data Types

8 bits: maximum representable value $2^8-1 = 255$

16 bits: maximum representable value $2^{16}-1 = 65,535$

In order to represent a negative value, a bit must be removed from the byte and used as a flag to indicate whether the value is positive or negative.

Integer Overflows

Signed 16 bit integer
−32,768 to 32,767

0111 1111 1111 1111 = 32,767
    +1
1000 0000 0000 0000 = -0 or 32,768
    +1
1000 0000 0000 0001 = -1 or 32,769
    +1
1000 0000 0000 0010 = -2 or 32,770

Integer Overflows

Some programs may make the assumption that a number always contains a positive value. If the number has a signature bit at the beginning, an overflow can cause its value to become negative.

An overflow may violate the program's assumption and may lead to unintended behavior.

Null Pointers

Null Pointers

Each application has its own address space, with which it is free to do with it as it wants.

NULL can be a valid virtual address in your application using mmap().

user land - uses virtual addresses

kernel land - uses physical addresses

Null Pointers

Address space switching expensive so the kernel just runs in the address space of whichever process was last executing.

At any moment, the processor knows whether it is executing code in user mode or in kernel mode.

Pages in virtual memory have flags on it that specifies whether or not user code is allowed to access it.

Null Pointers

Find kernel code that is initialized to NULL

```
struct my_ops {
    ssize_t (*do_it)(void);
};
static struct my_ops *ops = NULL;
```

This structure must be executed by the kernel

```
return ops->do_it();
```

Null Pointers

Disable OS security

```
echo 0 > /proc/sys/vm/mmap_min_addr
```

In your userland application you must declare code that is to be executed with kernel privileges

```
void get_root(void) {
    commit_creds(prepare_kernel_cred(0));
}
```

Map a page at zero virtual address

```
mmap(0, 4096, PROT_READ | PROT_WRITE , MAP_PRIVATE |
MAP_ANONYMOUS | MAP_FIXED , -1 , 0);
```

Null Pointers

Immediately Declare a pointer at null

```
void (**fn)(void) = NULL;
```

Set the address of our malicious userspace code as the value of our pointer

```
*fn = get_root;
```

Finally trigger the vulnerable kernel function that executes the structure. This is usually a syscall such as read(), write() etc...

Null Pointers

This works for 2 reasons:

(1.) - Since the kernel runs in the address space of a userspace process, we can map a page at NULL and control what data a NULL pointer dereference in the kernel sees, just like we could for our own process!

(2.) - To get code executing in kernel mode, we don't need to do any trickery to get at the kernel's data structures. They're all there in our address space, protected only by the fact that we're not normally able to run code in kernel mode.

Null Pointers

When the CPU is in user mode, translations are only effective for the userspace region of memory, thus protecting the kernel from user programs.

When in kernel mode, translations are effective for both userspace and kernelspace regions, thus giving the kernel easy access to userspace memory, it just uses the process' own mappings.

Without any exploit, just triggering the syscall would cause a crash.

Return Oriented Programming Chains (ROP)

ROP Chains

ROP chains are chains of short instruction sequences followed by a return instruction. These short sequences are called gadgets. Each gadget returns to the next gadget without ever executing a single bit from our non-executable shellcode. ROP chains enable attackers to bypass DEP and ASLR.

Since we cannot execute our own code on the stack, the only thing we can do is call existing functions. To access these existing functions using buffer overflow, we require at least one non-ASLR module to be loaded.

ROP Chains

Scan executable memory regions of libraries using automated tools to find useful instruction sequences followed by a return.

Existing functions will provide us with some options:
(1.) - execute commands (classic "ret-to-libc")
(2.) - mark memory as executable

A single gadget could look like this:

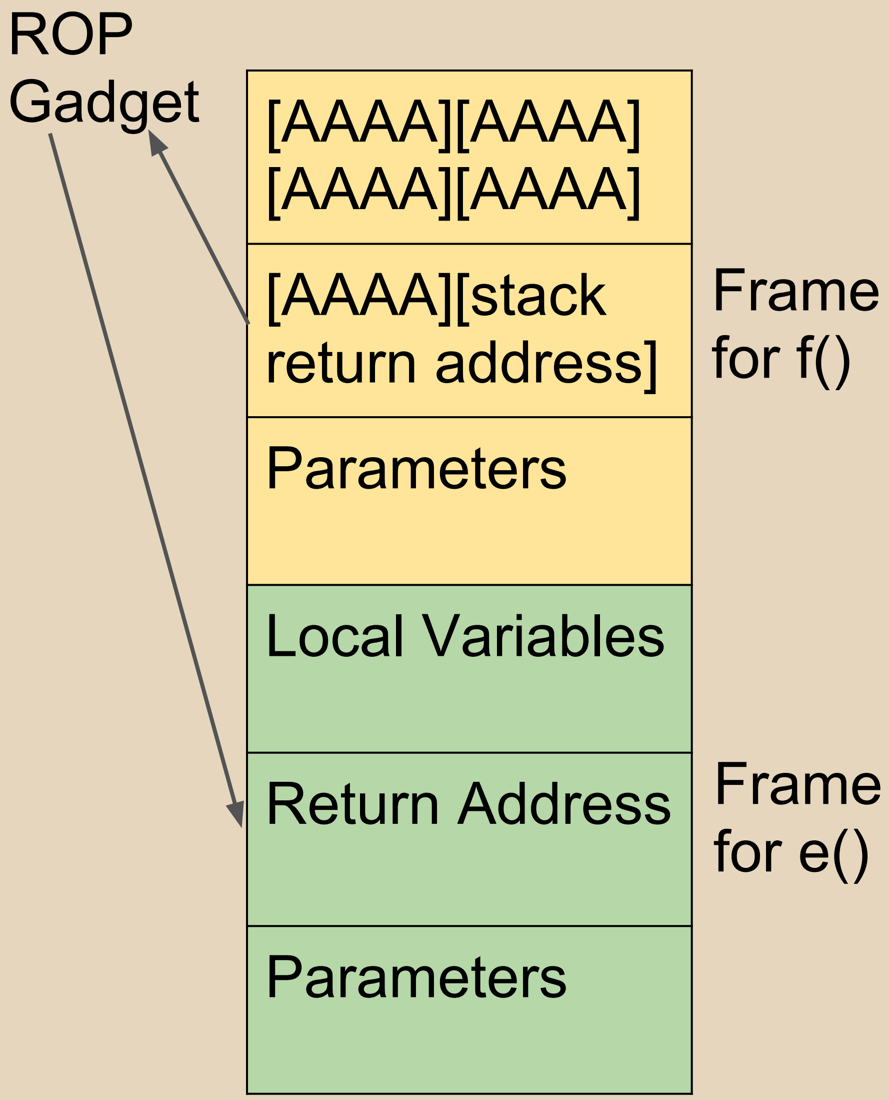
```
POP EAX
RET
```

ROP Chains

A single gadget would just continue to follow a path of execution that is being stored in the stack.

This means we only control the instruction pointer for 2 instructions.

We have to select a gadget that can alter multiple frames.



ROP Chains

Using a stack pivot, the ESP register (stack pointer) is loaded with the address to our own data so that input data is re-aligned and can be interpreted as return addresses and arguments to the called functions. Pivots basically enables us to use a forged stack.

Sample ROP exploit:

(1.) - Start the ROP chain

(2.) - Use a gadget pivot to the stack pointer to buffer

(3.) - Return to fake stack, and launch more gadgets

(4.) - Use gadgets to set up stack/registers

(5.) - Use gadgets to disable DEP/ASLR

(6.) - Return to shellcode and execute

Fin

Questions

Questions

References

SEH Exploit
https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

Heap Overflow
http://www.thehackerslibrary.com/?p=872
http://etutorials.org/Networking/network+security+assessment/Chapter+13.+Application-Level+Risks/13.5+Heap+Overflows/
http://geekscomputer.blogspot.com/2008/12/buffer-overflows.html
http://drdeath.myftp.org:881/books/Exploiting/Understanding.Heap.Overflow.Exploits.pdf
https://rstforums.com/forum/62318-run-time-detection-heap-based-overflows.rst
http://www.phrack.org/archives/57/p57_0x08_Vudo%20malloc%20tricks_by_MaXX.txt
http://www.cgsecurity.org/exploit/heaptut.txt
http://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf

Null Dereference Exploits
http://www.computerworld.com.au/article/212804/null_pointer_exploit_excites_researchers/
http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html
https://blogs.oracle.com/ksplice/entry/much_ado_about_null_exploiting1
http://blog.mobiledefense.com/2012/11/analysis-of-null-pointer-dereference-in-the-android-kernel/
http://lwn.net/Articles/342330/
http://lwn.net/Articles/75174/
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory
http://blog.mobiledefense.com/2012/11/analysis-of-null-pointer-dereference-in-the-android-kernel/

ROP Chains
https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/
https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/
http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf
http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf
http://www.exploit-monday.com/2011/11/man-vs-rop-overcoming-adversity-one.html
https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/
http://neilscomputerblog.blogspot.ie/2012/06/stack-pivoting.html