Stack-based BOF exploitation & Protection schemes (& how to break them)

Andrew Wesie & Brian Pak

Carnegie Mellon University

Plaid Parliament of Pwning

Who are we?

- Students at Carnegie Mellon University
- Hackers
  - Members of Plaid Parliament of Pwning

- ~~Codegate Participants~~
- Codegate Winners

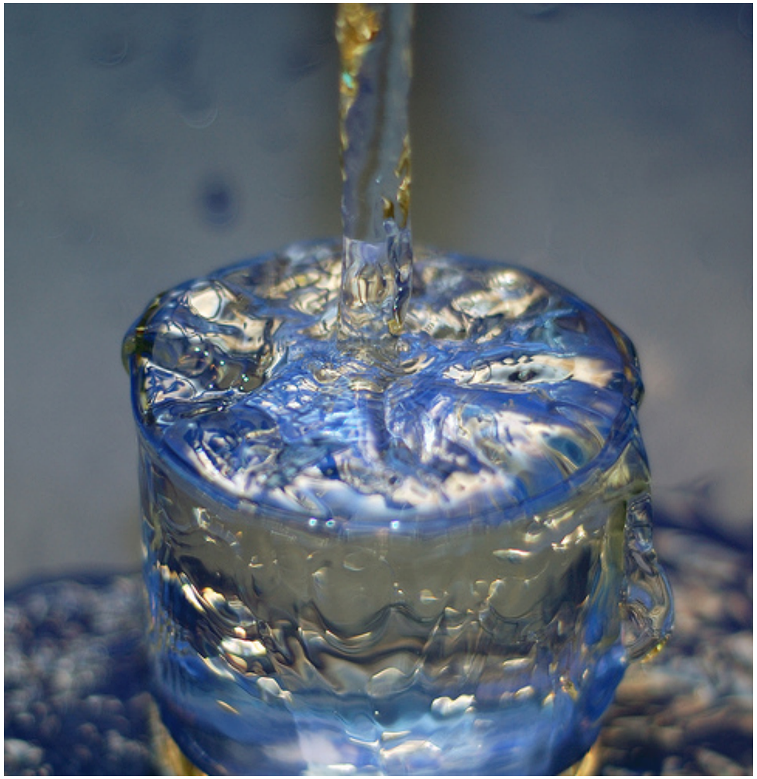
Agenda

- Buffer Overflow
- Exploitation techniques for stack-based buffer overflow
- Mitigation techniques to prevent the exploitation
- Bypassing the protections
- Summary
- Q&A

Buffer Overflow

& Exploitation

What is ‘Buffer Overflow’?



- Buffer overflow occurs when there is more data copied into the buffer than the size of the memory that is allocated for that buffer

```
Int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
}
```

What is ‘Buffer Overflow’?

- char buf[4];
- strcpy(buf, “plaid”);

p	l	a	i	d	\0

buf is only 4 bytes, but we copied total of 6 bytes of data (including NULL byte at the end)

Buffer Overflow

- There are two types of buffer overflow:
  - Stack buffer overflow
  - Heap buffer overflow
- It depends on where the overflow is happening
  - The cause is the same, but the way to attack is very different
- In this talk, we will only focus on Stack-based bof

What can happen?

- Segmentation Fault!
  - Programmer’s No. 1 enemy, Hacker’s No. 1 friend :)
- Due to the compiler allocating little more space than needed (for aligning), overflowing a couple of bytes wouldn’t cause a serious problem
- However, two things are clear with arbitrary # of bytes:
  - We can corrupt the local variable values
  - We can corrupt the saved ebp and *saved eip* on the stack

Assembly

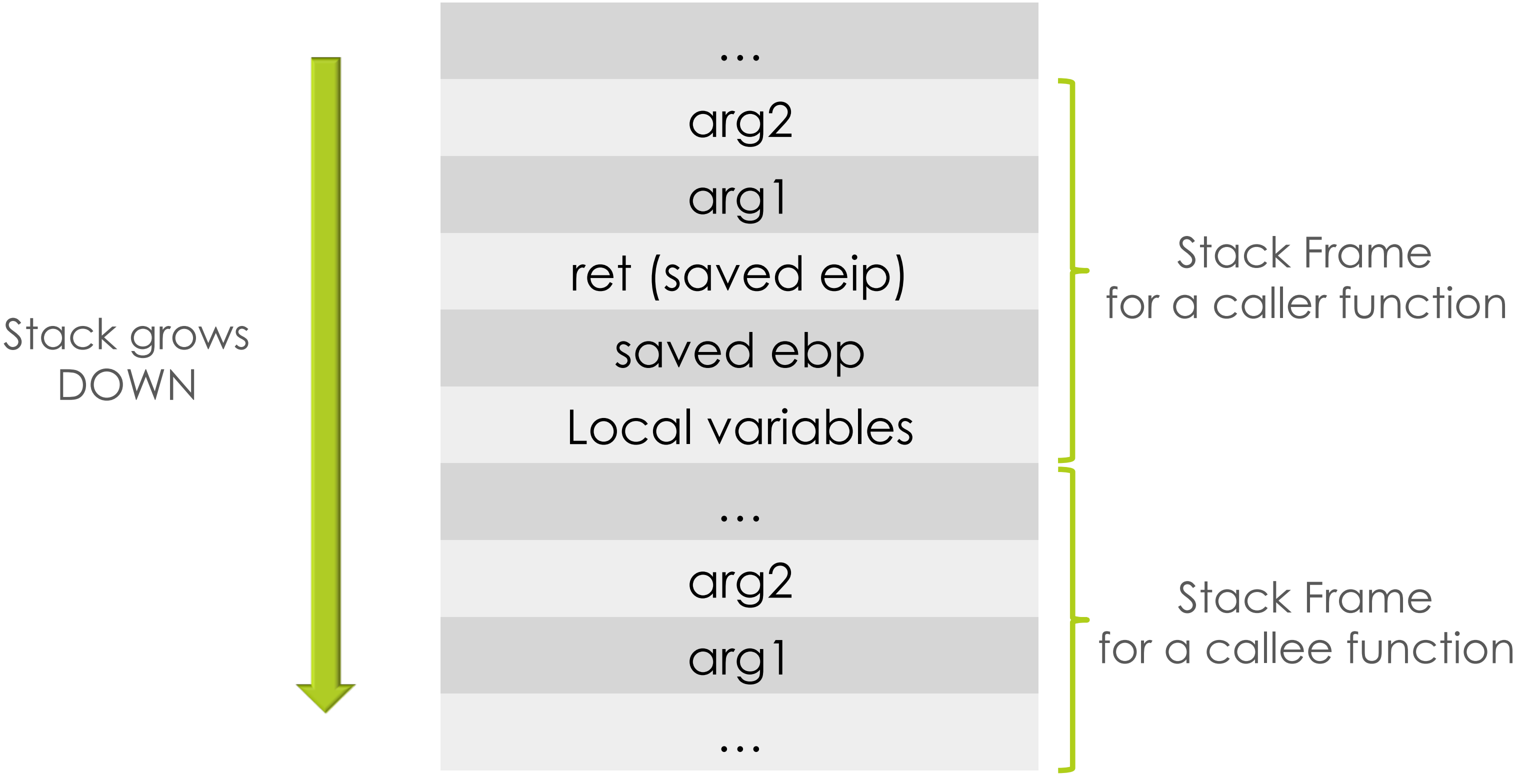
- In C:

```
int func(int arg1, int arg2)
{
    char buf[4];
        ...
    return 1;
}
```

- Disassembled:

```
<func>:
    push ebp              function
    mov ebp, esp          prologue
    sub esp, 0x8
        ...
    mov eax, 0x1          ; return value 1
    mov esp, ebp
    pop esp               function
    ret                   epilogue
```

Stack Layout



Stack grows DOWN

…
arg2
arg1
ret (saved eip)
saved ebp
Local variables
…
arg2
arg1
…

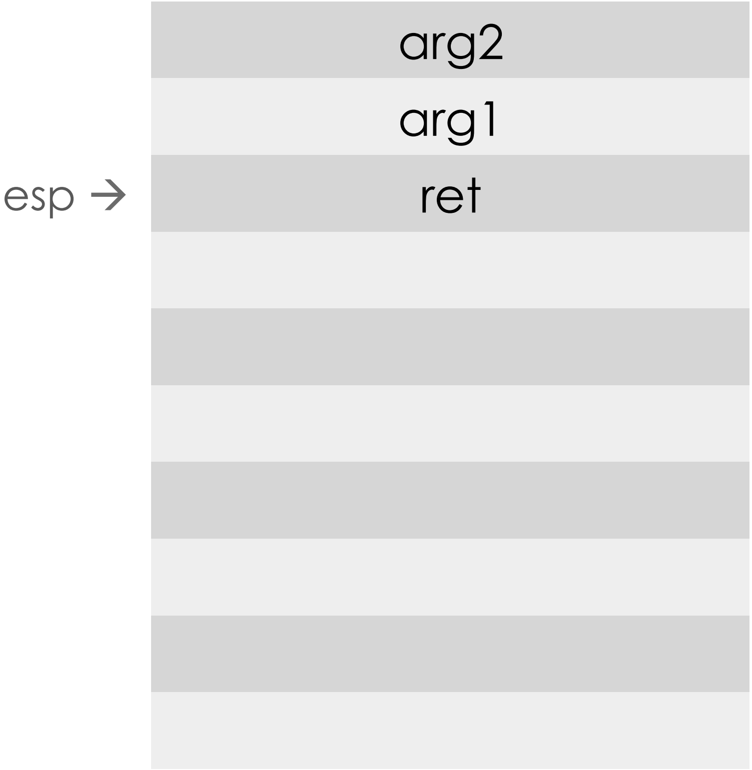
Stack Frame for a caller function

Stack Frame for a callee function

Stack Diagram

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      …
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```



Stack Diagram

- Code:

```
<func>:
    push ebp
    mov ebp, esp
    sub esp, 0x8
        ...
    mov eax, 0x1
    mov esp, ebp
    pop ebp
    ret
```

Stack Diagram

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      ...
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```

Stack Diagram

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      ...
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```

Stack Diagram

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      ...
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```

Stack Diagram

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      ...
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```

Stack Diagram

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      …
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```

arg2

arg1

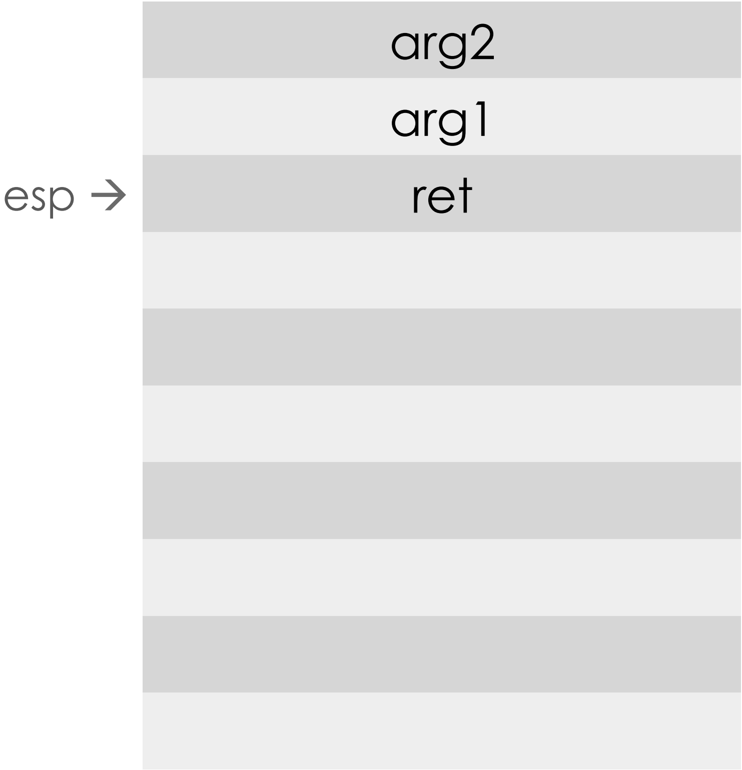
ret

ebp, esp → saved ebp

Stack Diagram

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      …
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```



arg2
arg1
esp → ret

ebp restored to ‘saved ebp’

Stack Diagram

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      …
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```

eip restored to ‘ret’

What if…

- In C:

```
int func(int arg1, int arg2)
{
   char buf[4];
         …
   strcpy(buf, …);
   return 1;
}
```

When copying 3 bytes…

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      …
   call strcpy
      …
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```

arg2

arg1

saved ret

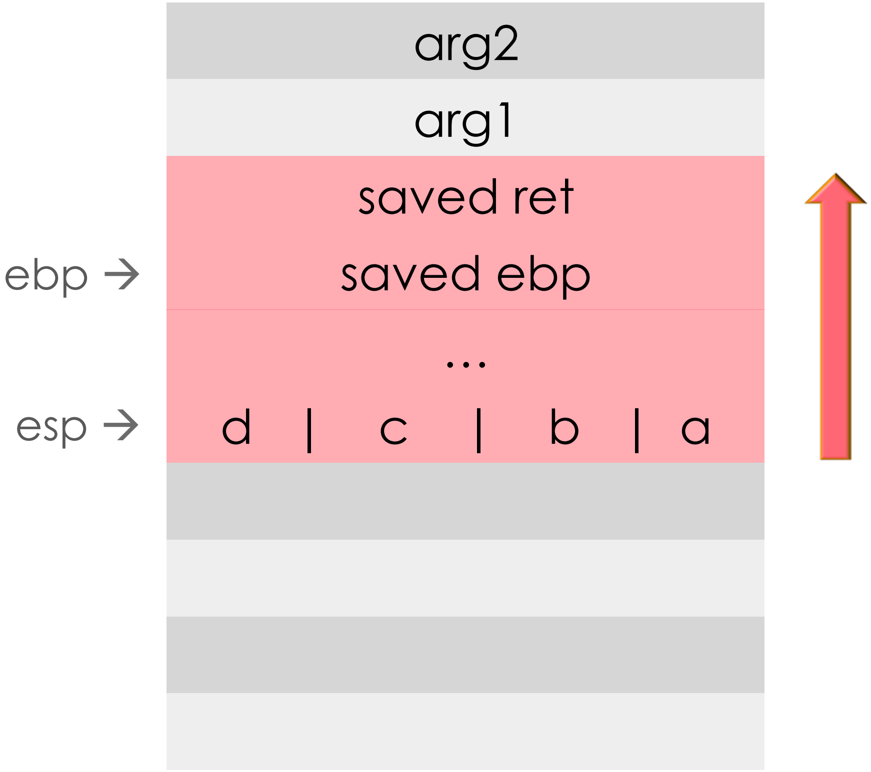
ebp → saved ebp

esp → \0 | c | b | a

When copying 16 bytes…

- Code:

```
<func>:
   push ebp
   mov ebp, esp
   sub esp, 0x8
      …
   call strcpy
      …
   mov eax, 0x1
   mov esp, ebp
   pop ebp
   ret
```



arg2

arg1

saved ret

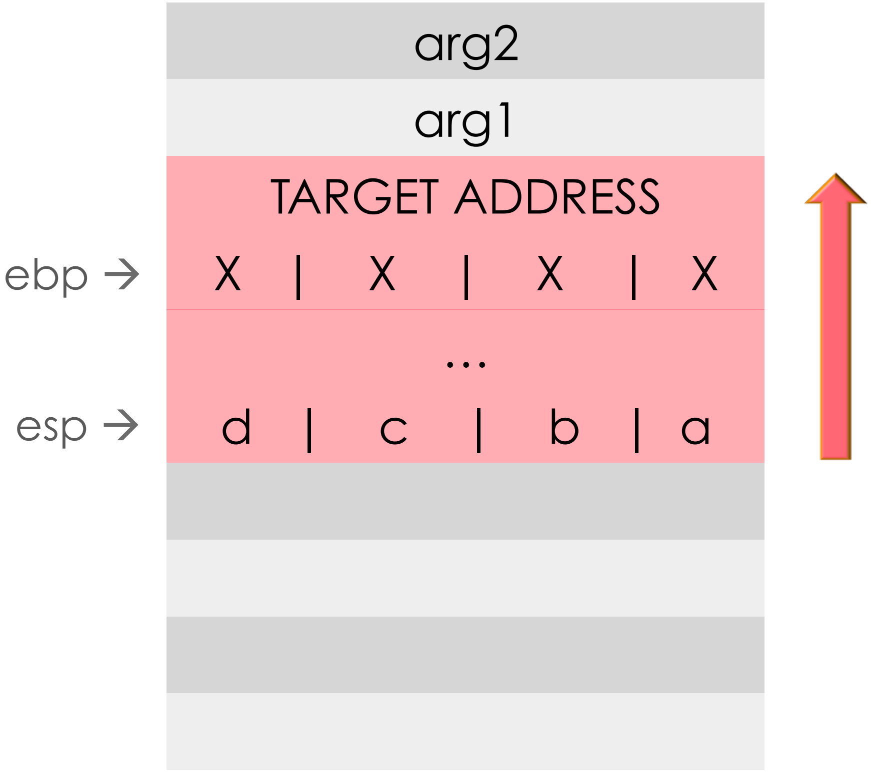
ebp → saved ebp

…

esp → d | c | b | a

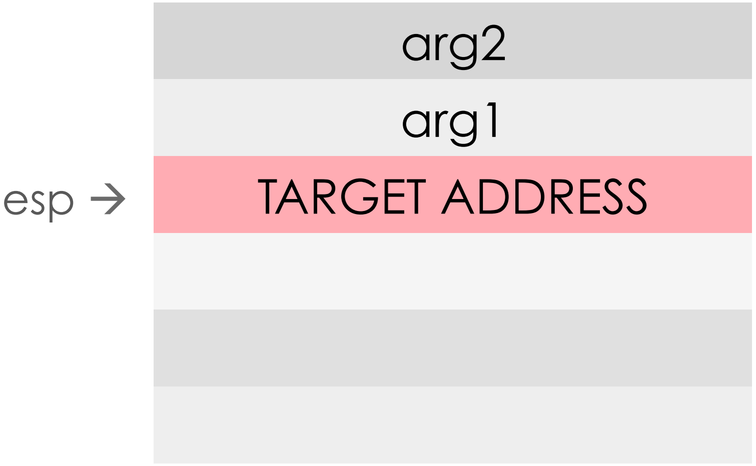
How it can be used to exploit

- Overflow the buffer such that we can modify the 'saved ret' to arbitrary target address



How it can be used to exploit

- Overflow the buffer such that we can modify the 'saved ret' to arbitrary target address
- When we reach the function prologue to return, we return to the target address that we set previously



arg2

arg1

esp → TARGET ADDRESS

```
mov esp, ebp
pop ebp
ret
    eip := [esp]
    esp += 4
```

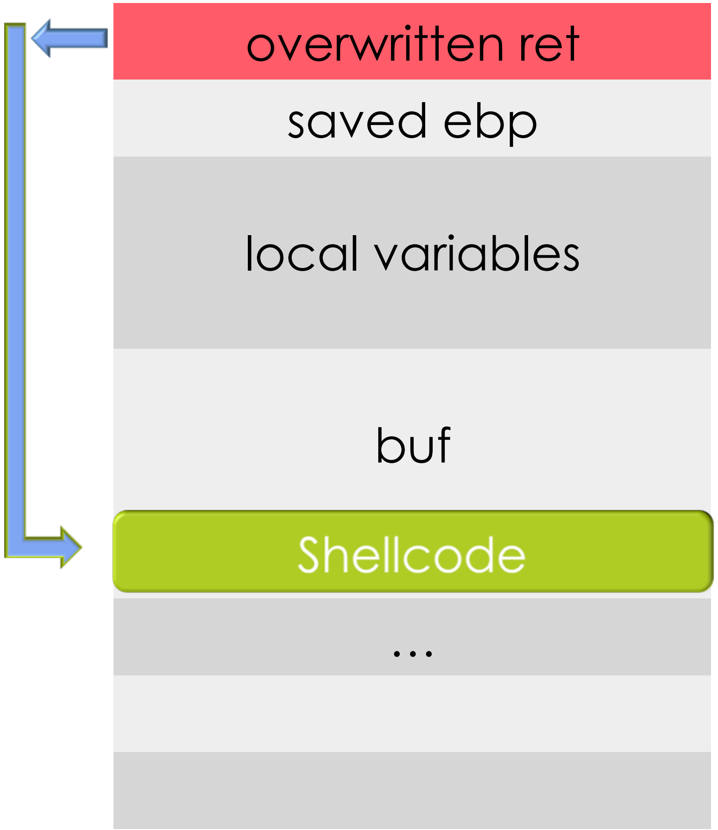
What does this mean to us?

- We can put arbitrary address to ‘saved ret’ and when the function returns, it will return to whatever we put
- If we put an address of an attacking code (e.g. shellcode), then that code will be executed
- In other words, we can control the flow of the program!

Demo I: Stack-based BOF

- Simple C Program
- Overwriting return address
  - Calling the function that is not called anywhere in the code
  - Executing Shellcode that's on the stack

Basic Idea



overwritten ret

saved ebp

local variables

buf

Shellcode

…

Protection Schemes

& How to bypass them

The Problem

- Remote code execution exploits need to be stopped
- It is hard to fix <u>all</u> bugs in <u>all</u> programs
- And it would be nice to make programs secure without re-compiling them

The Solutions

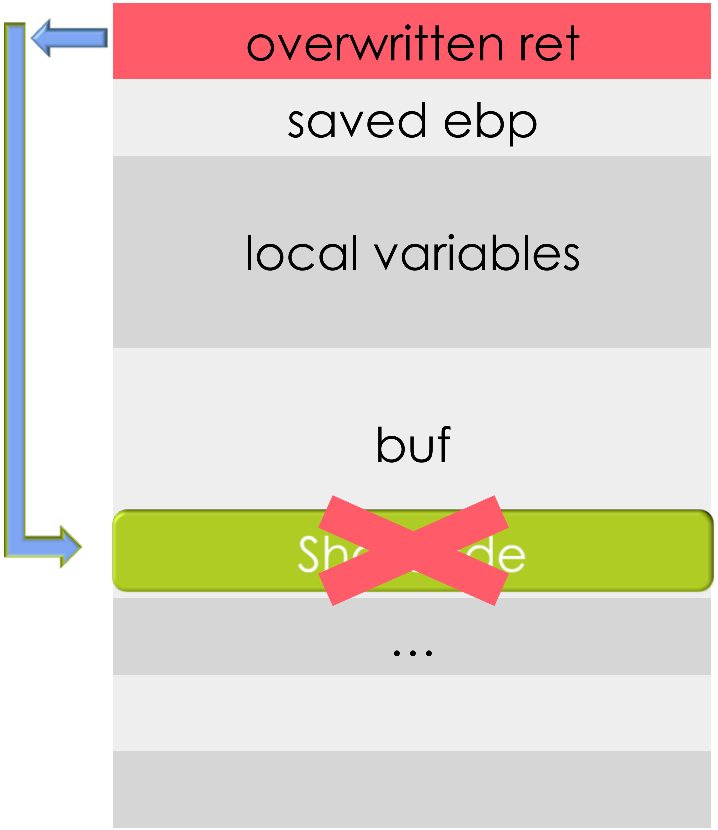
- Non-Executable Memory
  - NX-bit

- Randomization
  - ASLR
  - Stack Canary

NX-bit

- Originally, buffer overflows would execute code that the attacker provided
- So, can we never execute the attacker's code?
- Most of the operating systems support NX-bit, and is on by default.
- Basic idea:
    - Code is general read-only

Basic Idea



overwritten ret

saved ebp

local variables

buf

Sh de

...

NX-bit

- While it makes a buffer overflow exploit more complicated, it is not a perfect solution
- NX-bit does not stop the attacker from executing code that already exist in memory
- Typical techniques to bypass NX-bit protection
  - Return-To-Libc (RTL)
  - Return Oriented Programming (ROP)

Return-To-Libc

- The C libraries are almost always loaded into memory and contains lots of useful code
- ‘mprotect()’ can change the permissions of the memory
  - Making the attacker’s code executable
- ‘execv()’ will load a program and execute its code

Demo II: Same but with NX-bit

- Basic buffer overflow we just saw earlier
- We will show the way how we can’t exploit it with the same method
- We will demonstrate Return-to-Libc (system) to show how libc is useful :p

Return-Oriented-Programming

- Another method to bypass NX-bit protection
- A bit more complicated than Return-to-Libc attack
  - We will not go into the details here
  - For more details, read
    - http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html
    - http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf
- Find 'gadgets' in the code that is in memory
- Chain these gadgets using returns or jumps
- If you find the right set of gadgets, you will have a turing complete language

Stack Canaries (Stack cookie)

- Put a random number between stack variables and the return address
- Before executing a ‘ret’, verify the integrity of the random number
    - If the number changed, then abort
- Goal: detect bof, and stop them from being exploited

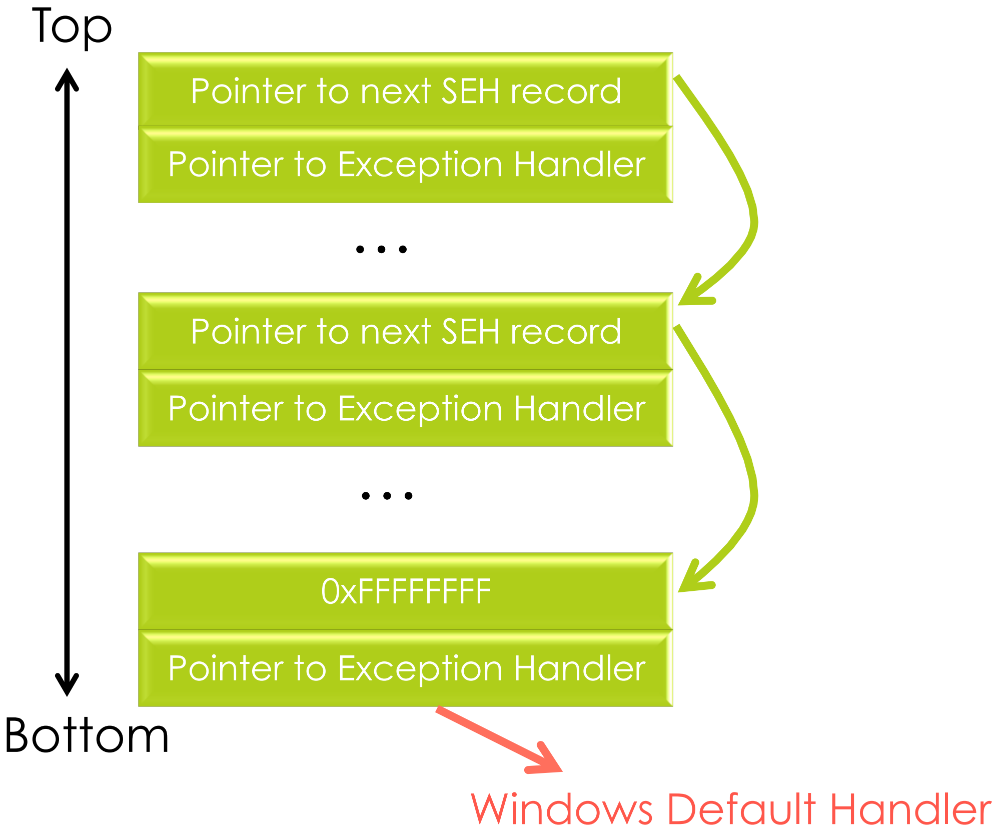
…
arg2
arg1
ret (saved eip)
saved ebp
STACK CANARY
Local variables

Stack Canaries

- Works very well on GNU/Linux
- On Windows, they can usually be bypassed with Structured Exception Handler (SEH) techniques
- Biggest flaw: they only protect stack
- It cannot stop things like:
  - Heap overflow / corruption
  - Double free
  - Format String Vulnerabilities

Structured Exception Handler

+ Stack View



Top

Pointer to next SEH record

Pointer to Exception Handler

…

Pointer to next SEH record

Pointer to Exception Handler

…

0xFFFFFFFF

Pointer to Exception Handler

Bottom

Windows Default Handler

- Mechanism to handle both hardware and software exceptions
- Supports __try, __except, and __finally keywords
- SEH frames saved on the stack
- In x86, FS register points to the current value of the Thread Information Block (TIB) structure
  - One element in TIB structure contains a pointer to an EXCEPTION_REGISTRATION structure.
- EXCEPTION_REGISTRATION structure points to the exception handler function

Structured Exception Handler

EXCEPTION_REGISTRATION structure

(EXSUP.INC in VC++ runtime library)

```
struct EXCEPTION_REGISTRATION

{

   EXCEPTION_REGISTRATION *prev;

   DWORD handler;

};
```

- Linked list
- prev points to the next EXCEPTION_REGISTRATION block
- handler contains a pointer to an exception handler function

Basic Concept

- We can overflow the buffer to overwrite data on the stack
- Then, we can overwrite SE Handler
  - Once the exception is handled, EIP will be changed to the address of the SE Handler
  - Thus, we can control the execution flow

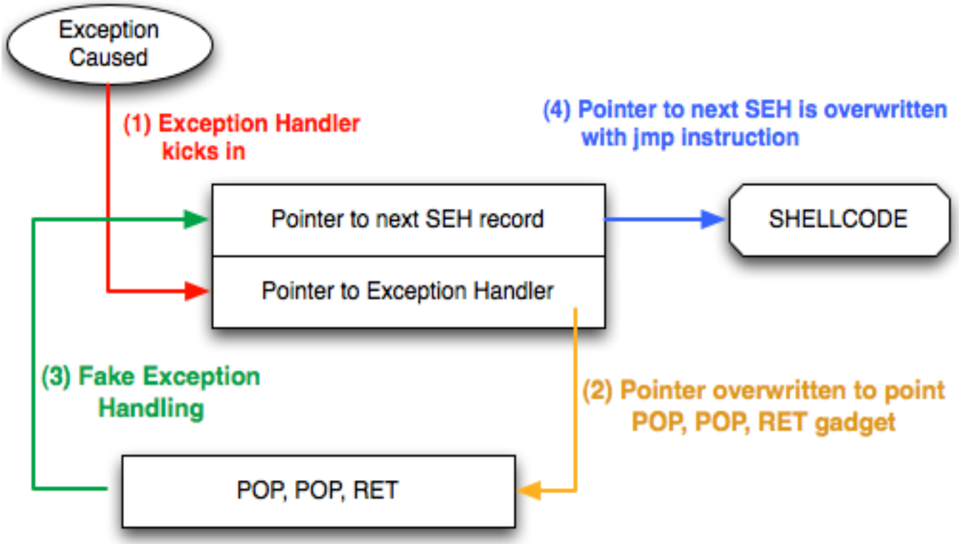
Wait…

- How is it useful if we have a stack canary, which will be verified later?

- SEH is awesome because:
    - If we can cause an exception before stack canary check occurs, it’s game over
        - And, we can:
            - Write beyond the end of the stack
    - Thus, no need to worry about stack canary being correct

SEH Exploit Design

- Overwrite the pointer to the next SEH record with jump instruction.
- Overwrite the SE Handler with a pointer to a sequence of instructions for fake exception handling.
- Cause an exception.
- Shellcode resides directly after the SE Handler.



*Redrawn from corelanc0der's tutorial

SEH Exploit Payload

- Usually, the SEH Exploit payload will be in the form of:
  - <Garbage> <next SEH> <SEH pointer> <Shellcode>
  - We put “jmp instruction” at <next SEH> to branch to <Shellcode>
  - We put the address of “pop, pop, ret” gadget at <SEH pointer>
    - This can be found from ntdll.dll or application specific dll’s
    - Check if there is any dll that is compiled without /SafeSEH

Demo III: Bypassing Stack Canary

- Windows
- Basic buffer overflow with /GS flag
  - We will show how it breaks the simple exploitation
  - We will show how SEH can be used to bypass this

Address Space Layout Randomization (ASLR)

- Trivial buffer overflow exploits rely on the location of the stack
- Return-to-Libc attacks (obviously) rely on the location of the libc in memory
- It is enabled in most of Linux and Windows distributions by default

Stack ASLR

- In a stack buffer overflow, this is easy to bypass
  - You can still overwrite the return address
  - Address of the attacker's buffer is usually on the stack
  - Otherwise, use Return-to-Libc or Return-Oriented-Programming
- In other scenarios, you will have to overwrite a function pointer instead of the return address
  - Entry in the GOT (Global Offset Table)
  - A virtual function table in C++

Library ASLR

- This was the answer to fix Return-to-Libc and Return-Oriented-Programming attacks
  - You cannot return to code if you don’t know where it is!
- In general, this does make life harder for the attacker
  - Not much in Windows though, since some libraries are not randomized
- Unfortunately, randomization might not be suffice

Other ASLR

- Heap is usually randomized if libraries are
  - This makes heap attacks more difficult
  - Usual work-around: heap spraying
- Program code can also be randomization
  - Rare in the real-world
  - Performance degrades, and have to enable at compile time
  - Position-independent code

Randomization Limitations

- Randomization is only effective if it stops the attacker from knowing the location of things

- Example: Randomization is useless if the attacker can combine buffer overflow with an information disclosure
    - If attacker can arbitrarily peek at memory before the overflow, he can figure out where things are ➔ making reliable exploits

Randomization Limitations

- Limited address space on x86
  - x86 has 32-bit address space
- Due to performance constraints, memory sections must be page-aligned
  - This reduces 32-bits of potential randomization to only 20-bits
- Libraries are located in a specific area of memory
  - Dependent on OS, distribution, etc.
  - Example: Debian=0xB7xxxxxx
  - This reduces library randomization by another 8 bits or so

Demo IV: Randomization Limits

- /proc /*/maps

-
  ```
  for i in `seq 1 4000`; do cat /proc/self/maps; done | grep
  “glibc line” | cut -f 1 -d ‘-’ | sort | uniq | wc -l
  ```
  - Returns 512 on Debian Squeeze 32-bit

Randomization Limitation

- If an attacker can attempt his exploit an arbitrary amount of times, then:
  - randomization becomes useless

- Main effect:
  - Exploits become less reliable
  - Attacks are now probabilistic

Demo V: Pwning NX and ASLR

- Show example with both NX and ASLR on
- Exploit the program using brute-force way (probabilistic method)

Other techniques

- These are the other techniques and topics that are related to this talk, but we haven’t covered them for the time being. Google them, and learn more about them!
  - Stack Pivot
  - Return Oriented Programming
  - Heap spray
  - Etc.

Conclusion

& some thoughts

Summary

- Buffer overflow
- Protection schemes against well-known attacks
  - NX-bit: You can’t run code in stack
  - Stack Cookie: You can’t overwrite the return address
  - ASLR: You don’t know the location of stack, heap, library
- Bypassing the protections
  - NX-bit: RTL (mprotect, execv), ROP, FSB, etc.
  - Stack Cookie: SEH overflow (Windows)
  - ASLR: Brute-forcing, Information Disclosure, etc.

Q&A

You will regret if you don’t ask it now!
(you can ask for my number too, if you want ;)