CSE 127: Computer Security

WI24

Lecture 5 - Memory safety and Isolation

Slide Credit: Kirill Levchenko, Stefan Savage, Stephen Checkoway, Hovav Shacham, David Wagner, Deian Stefan, Dan Boneh, and Zakir Durumeric , Nadia Heninger, George Obiado, Earlence Fernandes, Imani Munyaka

Announcements

Assignment 2 Released
Start Now - very serious

My OH on Calendar

Today

- Return-oriented programming
- Control flow integrity
- Heap corruption
- Isolation

Recall

- Recall: Data Execution Prevention (DEP/W^X)
  - Prevent attacker input (which is data) from being interpreted as code by marking data pages as non-executable
  - One exception: applications like browsers that explicitly mark some of their data as executable to Just-In-Time compilation (JIT)
  - Win!
- Is there another way for an attacker to execute arbitrary code even without the ability to inject it into the victim process?

Code Reuse Attacks

- Use the code that’s already there
- What code is already there?
  - – Program + shared libraries (including libc)

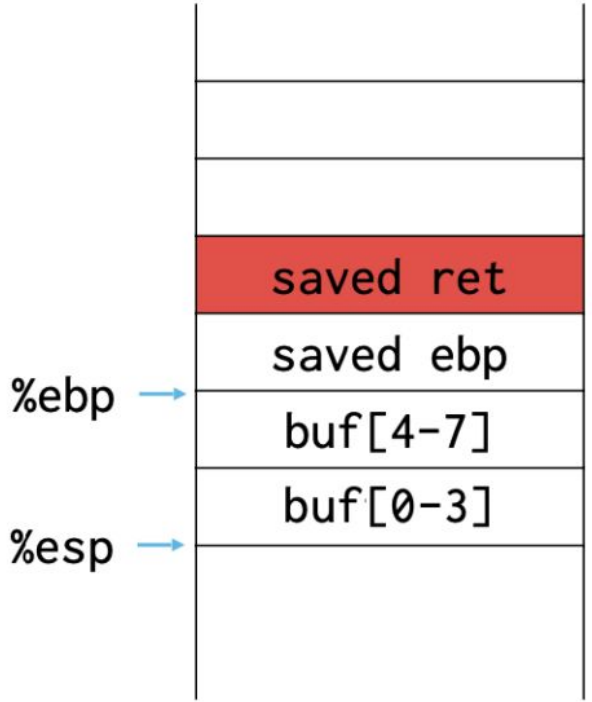
Return to Libc

- What can we find in libc?
  - “The system() library function uses fork(2) to create a child process that executes the shell command specified in command using execl(3) as follows:
    - execl("/bin/sh", "sh", "-c", command, (char *) 0);”
- Need to find the address of:
  - system()
  - String “/bin/sh”
  - Overwrite the return address to point to start of system()
  - Place address of “/bin/sh” on the stack so that system() uses it as the argument

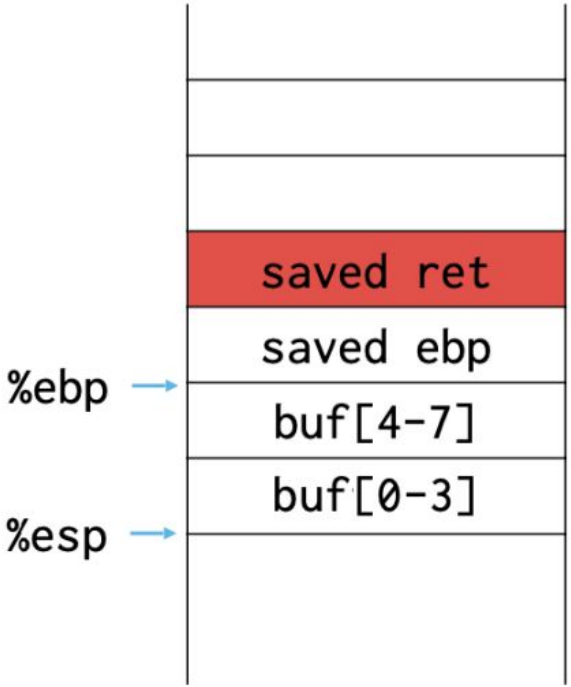
To be clean, you also want to push exit() on the stack so it will shut down gracefully

Redirecting control flow to system()

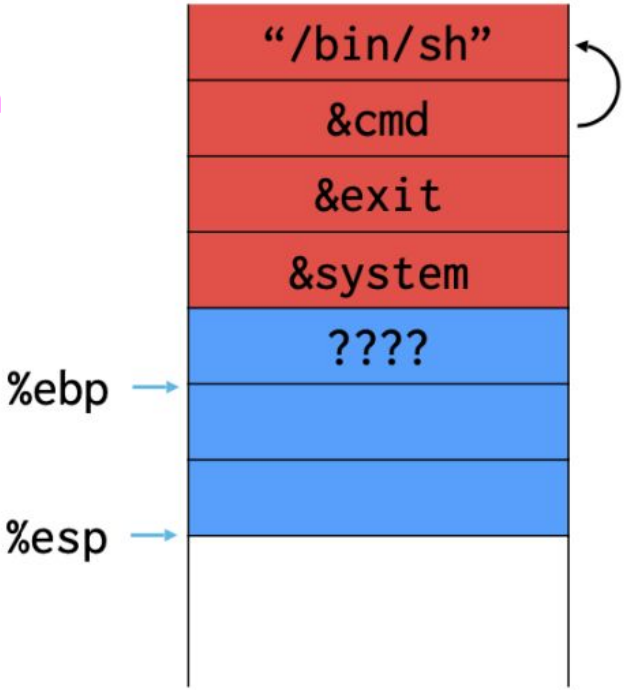
- We redirected control flow earlier to foo()
- Calling system() is the same, but need to have argument string “/bin/sh” on stack.



Redirecting control flow to system()



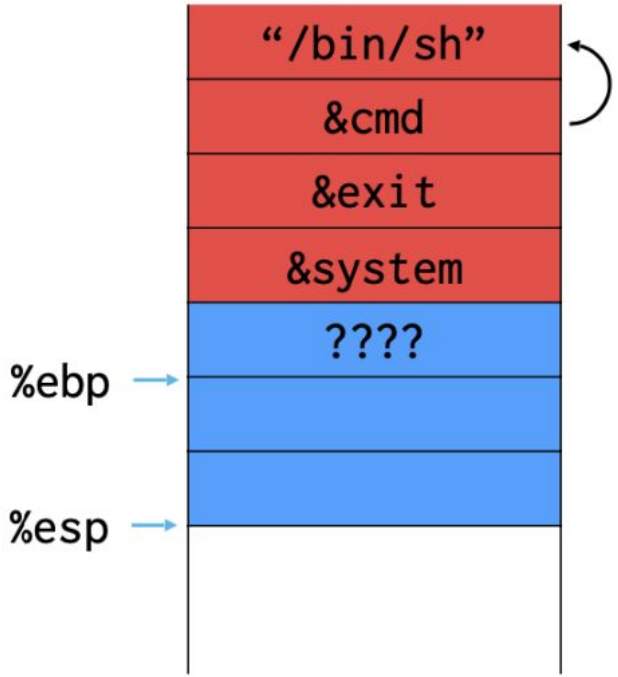
Remember: &system is the address of system(), which we can run because of our libc permissions →



Redirecting control flow to system()

leave → movl %ebp, %esp
pop %ebp

ret → popl %eip



Redirecting control flow to system()

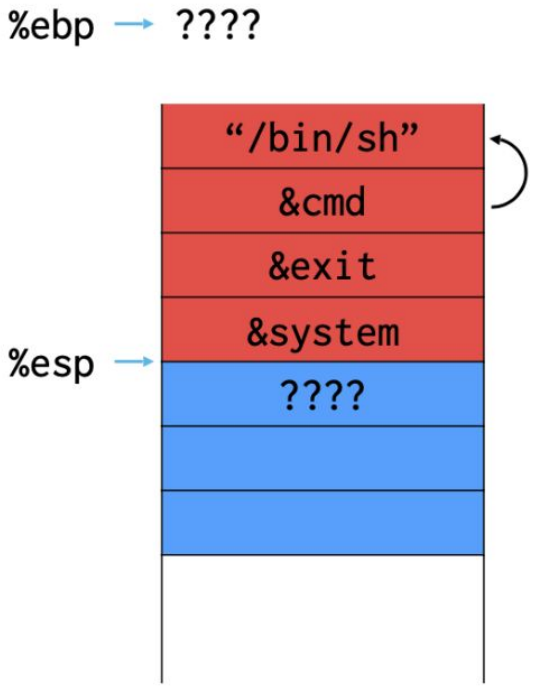
After leave

leave → movl %ebp, %esp
pop %ebp

→ popl %eip

ret

We're going to use ret to jump to system()



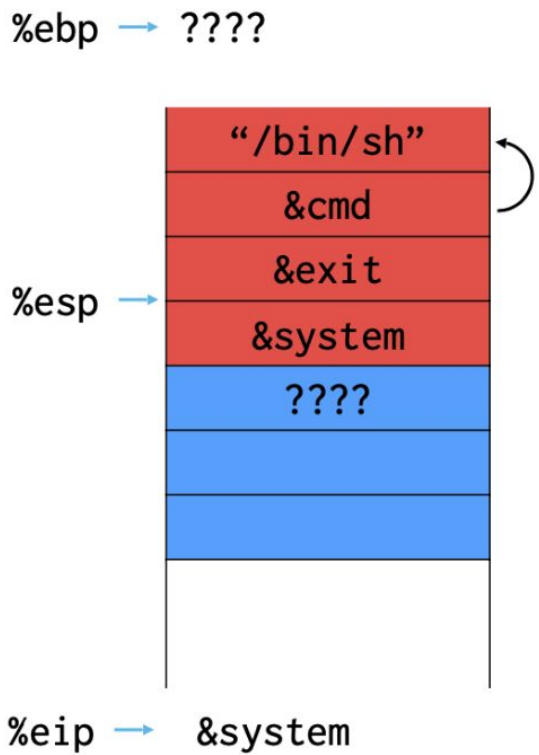
Redirecting control flow to system()

After ret

leave → movl %ebp, %esp
pop %ebp

ret → popl %eip

Executing system() gets arguments from stack

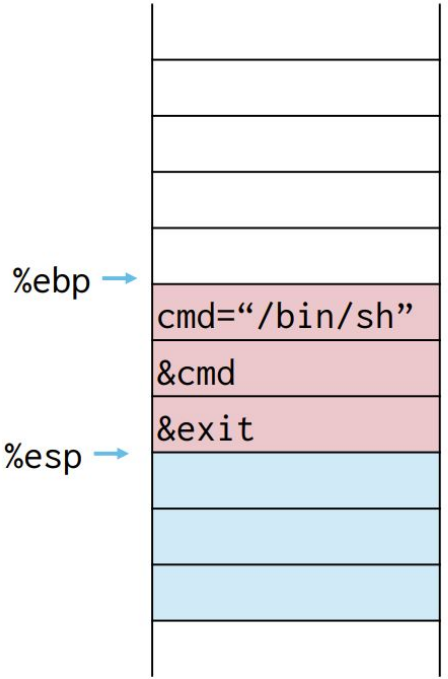


To system this looks like a normal call

Return-to-libc

What we want to get to

- Transfer control to address of system() in libc
- Setup stack frame to look like a normal call to system()
  - int system(const char *command);
  - &exit() system call is in the slot where the return address would be
  - &cmd is the argument
  - It points to the string “/bin/sh” stored further down the stack

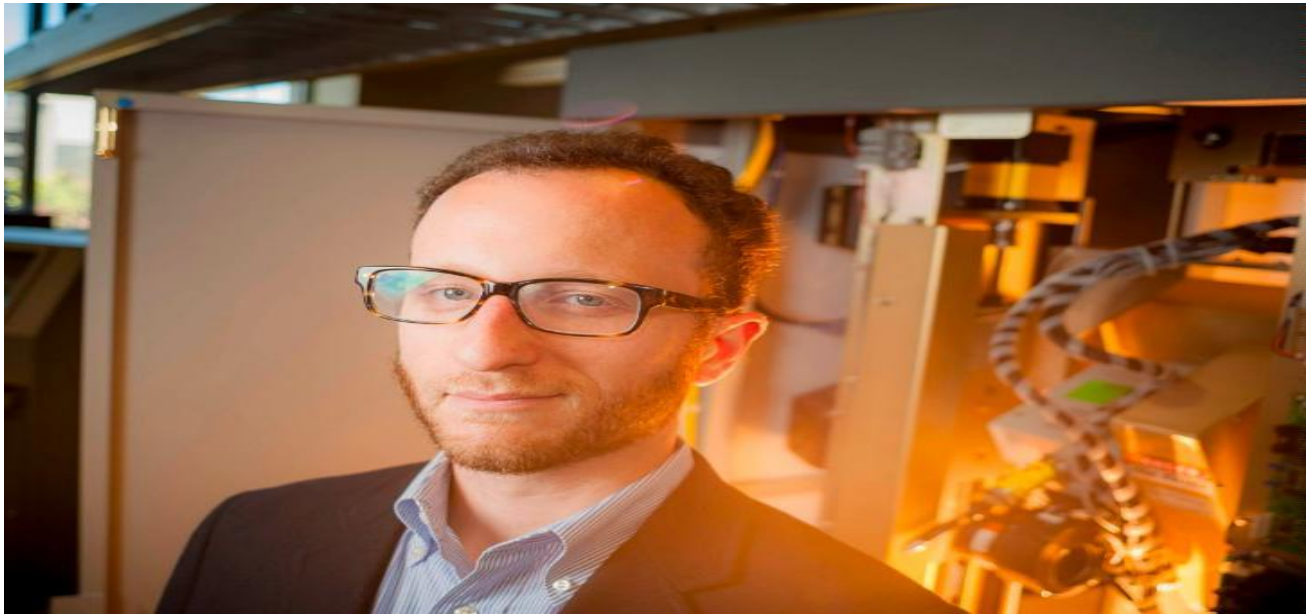


Return-to-libc is great, but...

what if there is no function that does what we want?

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu



Return-Oriented Programming

Idea: make shellcode out of existing code

Return-Oriented Programming

Idea: make shellcode out of existing code

Gadgets: code sequences ending in ret instruction

- Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.

Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.
- Where do you often find ret instructions?
  - End of function (inserted by compiler)
  - Any sequence of executable memory ending in 0xc3

Return-Oriented Programming

What happens if we jump almost to the end of some function?

– We execute the last few instructions, and then?

– Then we return. But where?

– To the return address on the stack. But we overwrote the stack with our own data so we control this address

- Let's choose to return to another tail of an existing function
- Rinse and repeat

x86 instructions

- Variable length!
- Can begin on any byte boundary!

Aside: how Intel variable length instructions help ROP

- X86 instructions are variable length, yet can begin on any byte address

- Example:

  81 c4 88 00 00 00 → add $0x00000088, %esp

  5f pop %edi
  5d pop %ebp
  c3 ret

- Result: more ”function tails” to choose from

Aside: how Intel variable length instructions help ROP

- X86 instructions are variable length, yet can begin on any byte address

- Example:

  81 c4 88 00 00 **00** → addb %bl, 93 (%edi)

  00 5f 5d ad db
  C3 ret

- Result: more ”function tails” to choose from

```
$ otool -t /bin/ls |grep c3
0000000100000f70        39 48 38 7f 07 b8 ff ff ff ff 7d 02 5d c3 48 83
0000000100000fc0        00 00 7d 02 5d c3 48 83 c6 68 49 83 c0 68 48 89
0000000100001010        c3 48 83 c7 68 48 83 c6 68 5d e9 6b 35 00 00 55
0000000100001050        b8 01 00 00 00 7d 02 5d c3 48 83 c6 68 49 83 c0
00000001000010a0        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 d8 34
00000001000010e0        48 7f 07 b8 01 00 00 00 7d 02 5d c3 48 83 c6 68
0000000100001120        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 58 34
0000000100001150        b8 01 00 00 00 7d 02 5d c3 48 83 c6 68 48 83 c1
00000001000011a0        7d 02 5d c3 48 83 c7 68 48 83 c6 68 5d e9 d8 33
00000001000011e0        58 7f 07 b8 01 00 00 00 7d 02 5d c3 48 83 c6 68
0000000100001870        c0 09 c8 8a 0d ab 3c 00 00 89 c3 81 cb 80 00 00
0000000100001b70        5d d4 89 de e8 57 29 00 00 48 89 c3 48 85 db 0f
0000000100001c30        03 39 00 00 01 e9 52 01 00 00 0f b7 c0 83 f8 0d
0000000100001dd0        c3 48 8d 35 91 2d 00 00 eb 07 48 8d 35 c0 2d 00
0000000100001e20        36 0f b7 56 58 83 fa 07 75 02 5d c3 44 0f b7 c9
0000000100001ec0        00 48 8d 3d e2 2c 00 00 e8 21 26 00 00 48 89 c3
0000000100001f70        34 48 83 c3 02 80 f9 3a 75 19 80 7b fe 3a 75 13
0000000100001fa0        c3 84 c9 75 d0 44 89 b5 78 fb ff ff 45 89 e6 80
00000001000023b0        fb ff ff 74 5c 8b 78 74 e8 ef 20 00 00 48 89 c3
00000001000023e0        00 00 48 89 c3 48 85 db 0f 84 9a 04 00 00 48 89
0000000100002520        66 18 4d 8b 7e 20 41 8b 5e 30 48 63 c3 48 8d 34
0000000100002560        20 49 63 4e 30 41 89 04 8f 41 8b 5e 30 ff c3 41
0000000100002870        38 05 00 00 5b 41 5c 41 5d 41 5e 41 5f 5d c3 48
0000000100002970        c3 48 8d 3d 9e 22 00 00 48 8d 35 a1 22 00 00 48
0000000100002a30        ed 48 83 c3 68 48 89 df e8 4f 0b 00 00 89 c3 45
0000000100002a90        0f b7 7c 24 04 e8 28 0b 00 00 01 c3 89 d8 48 83
0000000100002aa0        c4 08 5b 41 5c 41 5d 41 5e 41 5f 5d c3 55 48 89
0000000100002c30        4f 28 48 8b 46 08 eb 0f 85 c0 45 8b 4f 38 48 8b
0000000100003200        00 48 83 c3 18 48 81 fb a8 01 00 00 75 84 bb 10
0000000100003260        45 89 fd 4c 8d bd b0 f7 ff ff 48 83 c3 18 48 83
00000001000032e0        5b 41 5c 41 5d 41 5e 41 5f 5d c3 48 8d 35 c5 18
0000000100003350        48 83 c4 08 5b 5d c3 48 8d 3d c6 1b 00 00 31 c0
00000001000034a0        c4 70 5b 41 5e 5d c3 e8 a0 0f 00 00 55 48 89 e5
0000000100003550        00 89 d8 48 83 c4 08 5b 5d c3 66 90 7e ff ff ff
00000001000035f0        00 00 00 5d c3 81 c1 00 60 00 00 81 e1 00 f0 00
00000001000036a0        75 06 48 83 c3 10 eb 69 48 8d 7b 68 e8 f7 0e 00
0000000100003740        5e 41 5f 5d c3 55 48 89 e5 41 57 41 56 41 55 41
0000000100003930        5c f0 ff ff 89 c3 48 8d 05 1b 1d 00 00 8b 08 85
0000000100003970        7c 04 85 c9 75 40 41 89 d4 89 c3 48 8d 05 b6 1c
0000000100003990        45 f8 e8 c3 0b 00 00 42 8d 04 2b 23 45 c8 44 89
00000001000039f0        83 c4 38 5b 41 5c 41 5d 41 5e 41 5f 5d c3 31 ff
0000000100003ac0        00 00 48 89 c3 8a 04 1a 88 45 d6 48 83 ca 01 48
0000000100003b00        f8 80 f9 30 75 36 83 c3 d0 41 89 1f 66 bb 01 00
0000000100003b40        9f 80 f9 07 77 08 83 c3 9f 41 89 1f eb 4e 89 c1
0000000100003b50        80 c1 bf 80 f9 07 77 12 83 c3 bf 41 89 1f 48 8b
0000000100003bd0        41 5d 41 5e 41 5f 5d c3 55 48 89 e5 41 56 53 41
0000000100003c30        c6 08 00 00 89 c7 44 89 f6 5b 41 5e 5d e9 e2 08
0000000100003c60        31 c0 48 83 c4 10 5d c3 55 48 89 e5 e8 e9 08 00
0000000100003c70        00 31 c0 5d c3 55 48 89 e5 41 56 53 89 f8 48 8d
0000000100003d10        5e 5d e9 b5 08 00 00 5b 41 5e 5d c3 55 48 89 e5
0000000100003e40        ff ff 4c 89 e6 4c 89 f9 e8 f5 06 00 00 48 89 c3
0000000100003e90        98 00 00 00 5b 41 5c 41 5d 41 5e 41 5f 5d c3 e8
```

Return Oriented Programming

ROP idea: make shellcode out of existing application code.

- Stitching together arbitrary programs out of code gadgets already present in the target binary
  - ROP Gadgets: code sequences ending in ret instruction.
  - Commonly added by compiler (at end of function)
  - But also (on x86) any sequence in executable memory ending in 0xC3 (ret).
- x86 has variable-length instructions
- Misalignment (jumping into the middle of a longer instruction) can produce new, unintended, code sequences
- Overwrite saved return address on stack to point to first gadget, the following word to point to second gadget, etc

- Stack pointer is the new instruction pointer in this crazy world

Return Oriented Programming

▪ **“Our thesis:** In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.” – The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86) by Hovav Shacham – https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf ▪

Turing-complete computation
– Load and Store gadgets
– Arithmetic and Logic gadgets
– Control Flow gadgets

One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3 =

```
mov $0x1,%eax
pop %ebx
leave
ret
```

One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3 =

```
add %al,(%eax)
pop %ebx
leave
ret
```

One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3 = add %bl,-0x37(%eax)
ret

One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3 = pop %ebx
leave
ret

One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3 = leave
ret

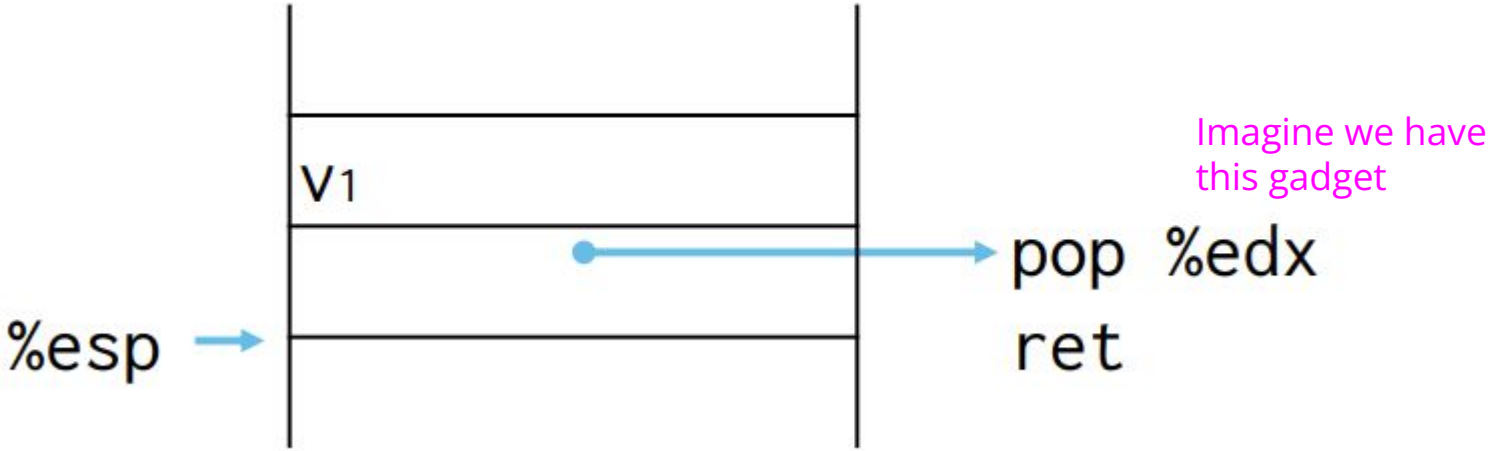
One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3 = ret

Why RET ?

- Attacker overflows stack allocated buffer
- What happens when function returns?
  - Restore stack frame
    - leave = movl %ebp, %esp;
      pop %ebp
  - Return
    - ret = pop %eip
- If instruction sequence at %eip ends in ret what do we do?

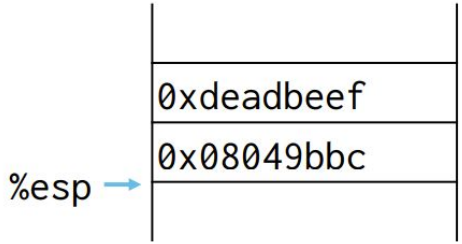
What happens if this is what we overflow the stack with?



relevant register(s):

%edx = 0x00000000

relevant stack:



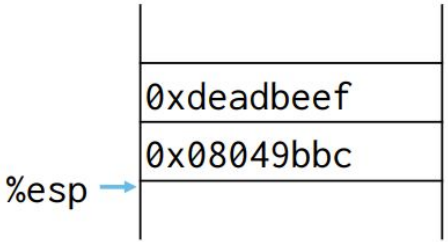
relevant code:

```
%eip → 0x08049b62: nop
       0x08049b63: ret
                   ...
       0x08049bbc: pop %edx
       0x08049bbd: ret
```

relevant register(s):

```
%edx = 0x00000000
```

relevant stack:



relevant code:

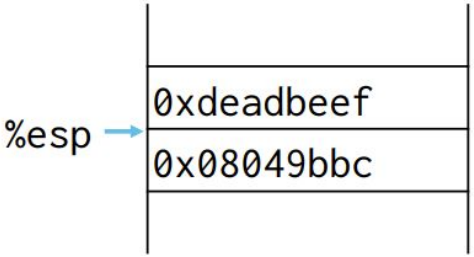
```
          0x08049b62: nop
%eip →    0x08049b63: ret
                ...

          0x08049bbc: pop %edx
          0x08049bbd: ret
```

Pop %edx

relevant register(s):

%edx = 0x00000000

relevant stack:



relevant code:

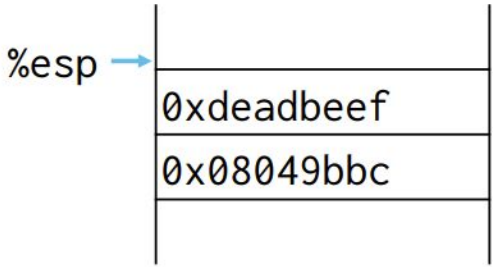
```
          0x08049b62: nop
          0x08049b63: ret
                      ...
%eip →    0x08049bbc: pop %edx
          0x08049bbd: ret
```

ret

relevant register(s):

%edx = 0xdeadbeef

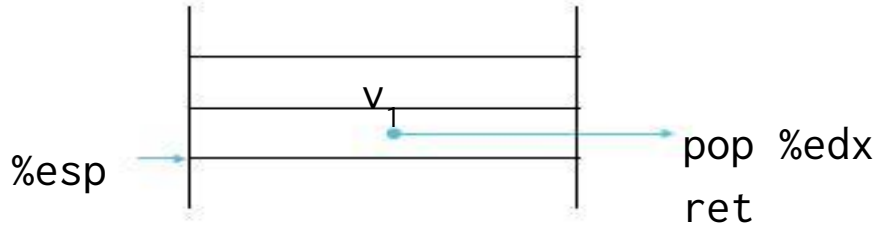
relevant stack:



relevant code:

```
            0x08049b62: nop
            0x08049b63: ret
                        ...
            0x08049bbc: pop %edx
%eip →      0x08049bbd: ret
```

This is a ROP gadget!



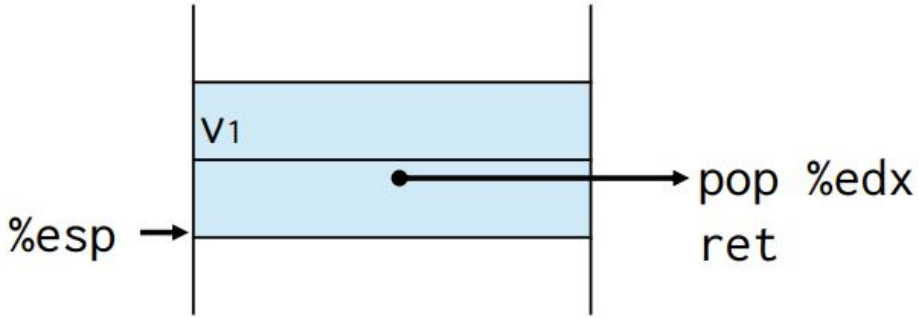
This gadget is the same as

```
movl v1, %edx
```

How do you use this as an attacker?

- Overflow the stack with values and addresses to such gadgets to express your program

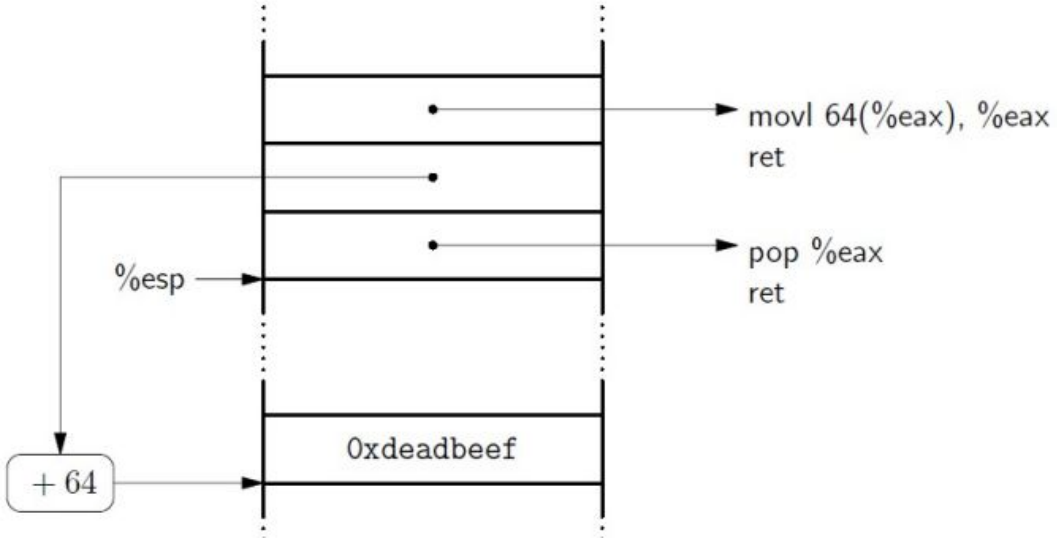
- e.g. if shellcode needs to write a value to %edx, use the previous gadget



Return Oriented Programming

Gadget for loading a value from memory

- A bit more complex…
- Attacker sets up stack so address of value to be loaded is on stack
  - Technically, 64 bytes less than addr
- Return to gadget that pops that address into %eax
- Return to gadget that loads the value based on address in %eax



Can express arbitrary programs

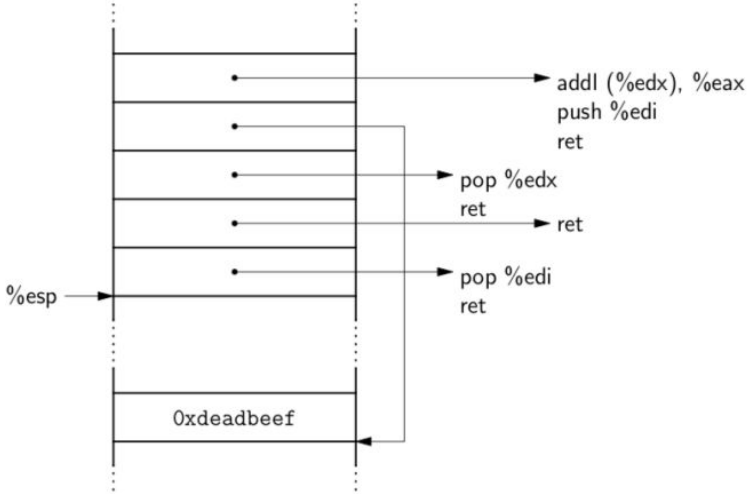


Figure 5: Simple add into %eax.

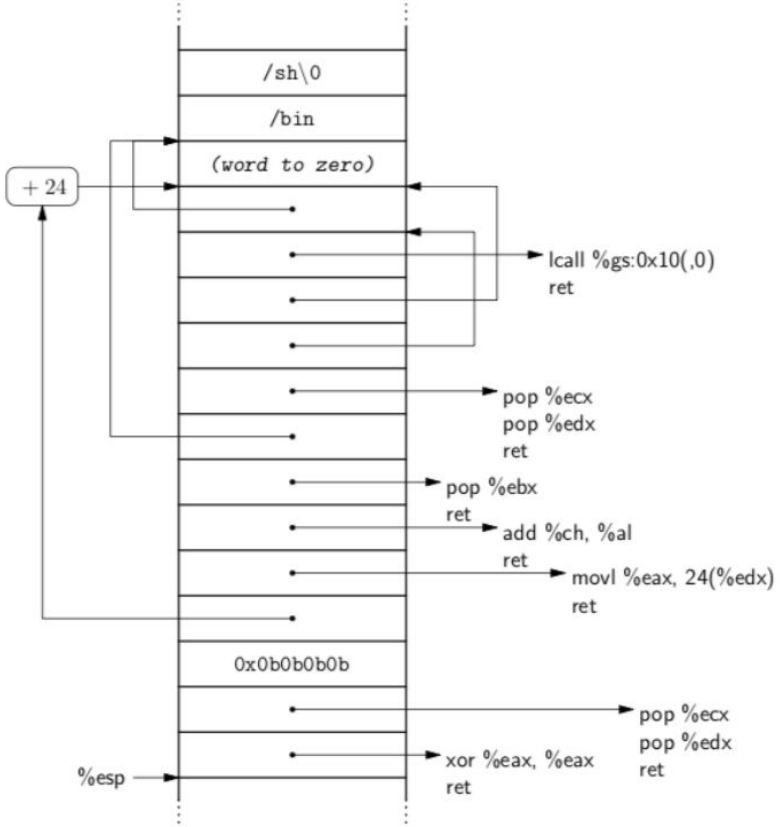


Figure 16: Shellcode.

Return Oriented Programming

Stack pointer acts as instruction pointer

- Manually stitching gadgets together gets tricky
  - Automation to the rescue!
  - Gadget finder tools, ROP chain compilers, etc.
  - All of this has been quasi-automated
- UCSD built first ROP compiler as grad class project in 2009
- As of 2013 CIA had a working ROP compiler toolchain (Vault7 leaks)
- Also: not even really about “returns”… other variants target other kinds of deterministic control flow (e.g. jump tables)
- Well, heck…. what to do?

Can find gadgets automatically

Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University

Ropper - rop gadget finder and binary information tool

You can use ropper to look at information about files in different file formats and you can find ROP and JOP gadgets to build chains for different architectures. Ropper supports ELF, MachO and the PE file format. Other files can be opened in RAW format. The following architectures are supported:

- x86 / x86_64
- Mips / Mips64
- ARM (also Thumb Mode)/ ARM64
- PowerPC / PowerPC64

How do you mitigate ROP?

Observation: In almost all the attacks we looked at, the attacker is overwriting jump targets that are in memory (return addresses and function pointers)

Today

- Return-oriented programming

→ Control flow integrity

- Heap corruption
- Isolation

Control Flow Integrity

- **Idea:** Don’t try to stop the memory writes.
- **Instead:** Restrict control flow to legitimate paths
  - Ensure that jumps, calls, and returns can only go to allowed target destinations

Control Flow Integrity

Focus is on protecting indirect transfer of control flow instructions.

- **Direct control flow transfer:**
  - Advancing to next sequential instruction
  - Jumping to (or calling a function at) an address hard-coded in the instruction
  - These are static in code, so assume attacker can't control (if they can overwrite code segment its game over anyway)

- **Indirect control flow transfer**
  - Jumping to (or calling a function at) an address in register or memory
  - Forward path: indirect calls and branches (e.g., a function you are calling)
  - Reverse path: return addresses on the stack (returning from a called function)

Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e. direct jumps and calls)?

Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e. direct jumps and calls)?
  - Address is hard coded in instruction. Not under attacker control.

Restricting indirect transfers of control

What are the ways to transfer control indirectly?

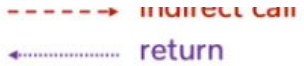
- **Forward path:** Jumping to or calling a function at an address in register or memory
  - e.g. qsort, interrupt handlers, virtual calls, etc.
- **Reverse path:** Returning from function using address on stack

What's a legitimate target?

Look at the program control-flow graph!

```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```

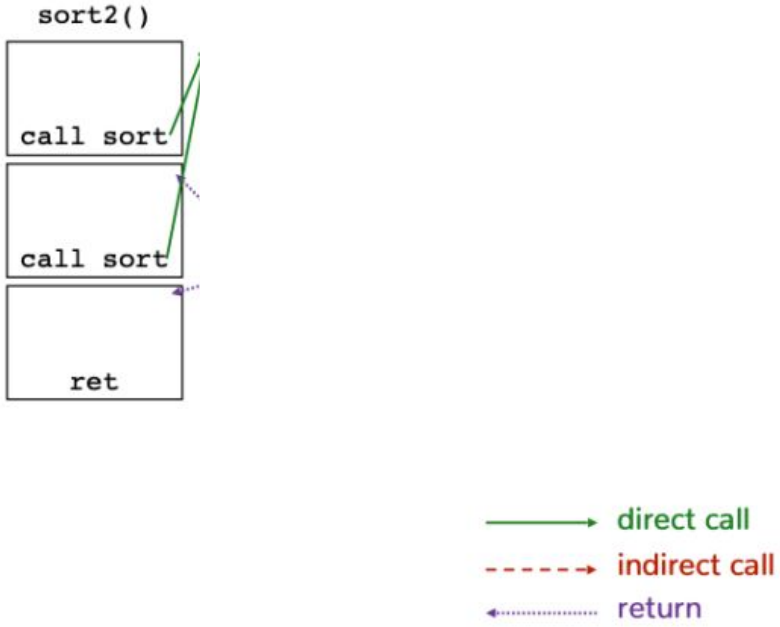
A control-flow graph is representation, using graph notation, of all paths that might be traversed through a program during its execution. (wikipedia.com)



What's a legitimate target?

Look at the program control-flow graph!

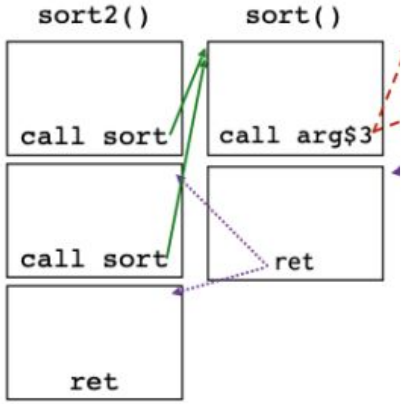
```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```



What's a legitimate target?

Look at the program control-flow graph!

```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```

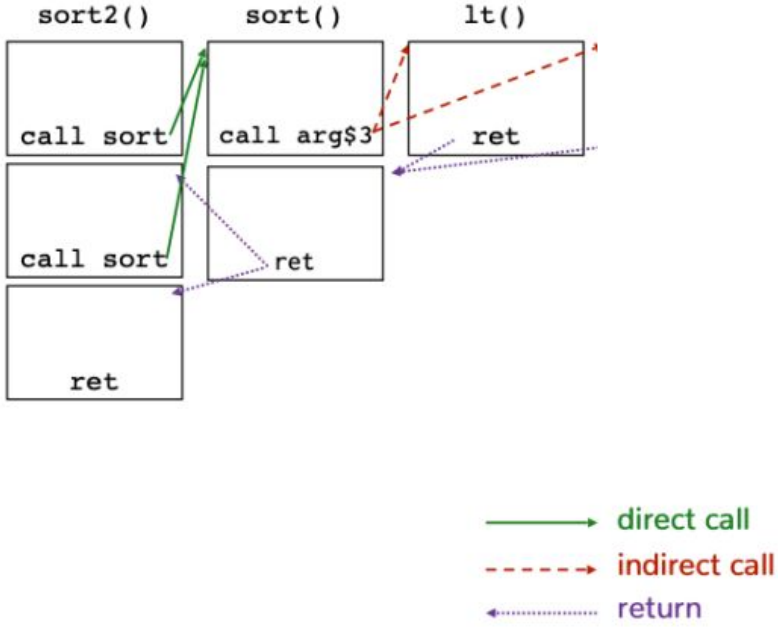


direct call
indirect call
return

What's a legitimate target?

Look at the program control-flow graph!

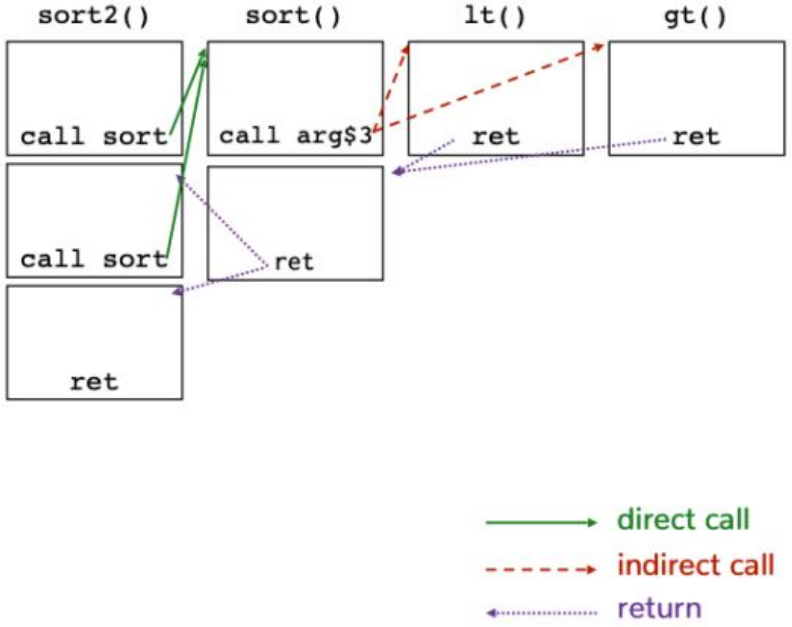
```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```



What's a legitimate target?

Look at the program control-flow graph!

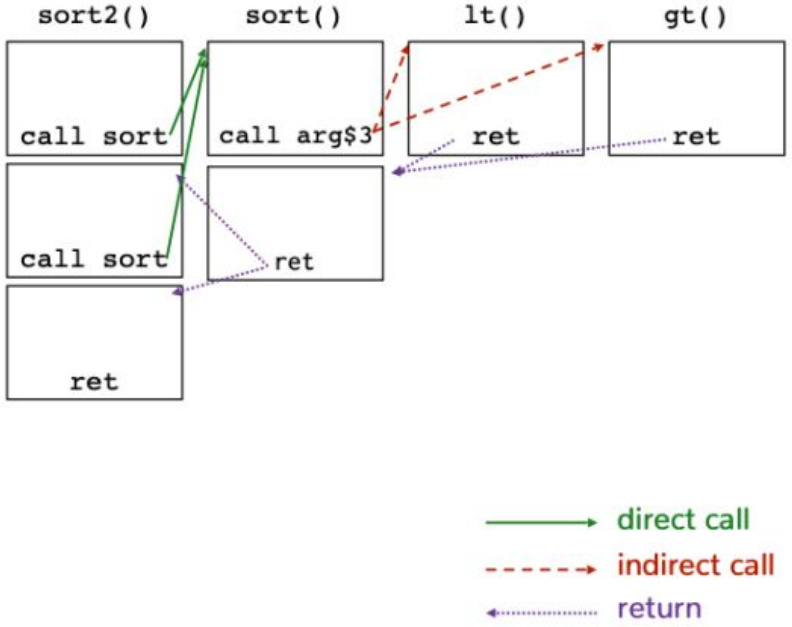
```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```



What's a legitimate target?

Look at the program control-flow graph!

```
void sort2(int a[],int b[], int len {
    sort(a, len, lt);
    sort(b, len, gt);
}
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
```

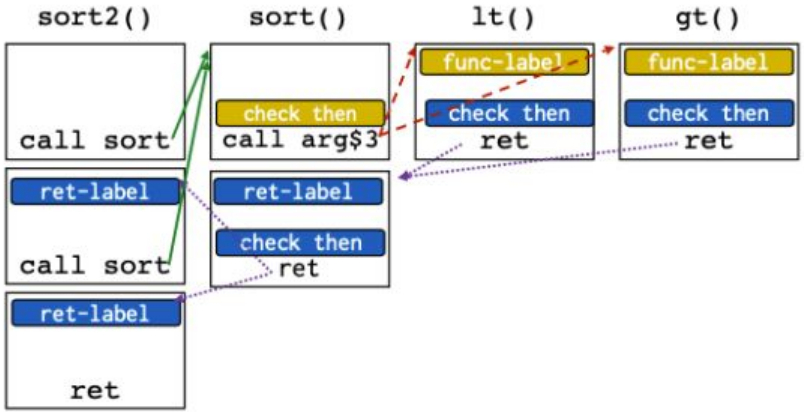


How do we restrict jumps to control flow graph?

- Assign labels to all indirect jumps and their targets
- Before taking an indirect jump, validate that target label matches jump site
  - Like stack canaries, but for control flow target
- Need hardware support
  - Otherwise trade off precision for performance

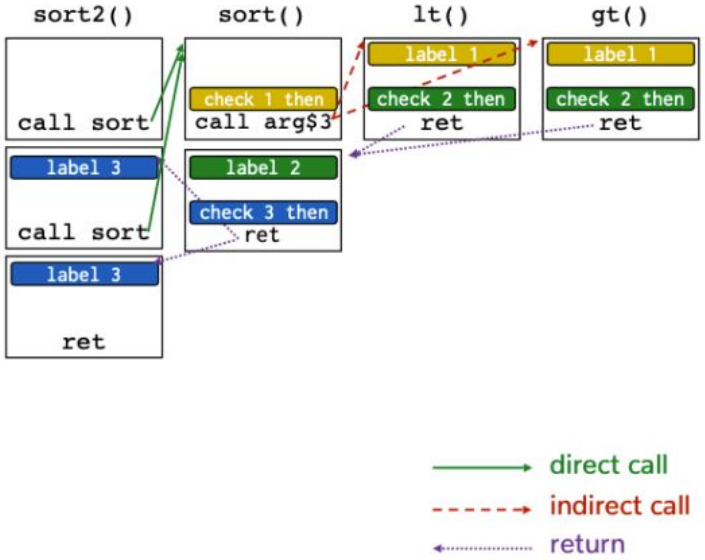
Coarse-grained CFI (bin-CFI)

- Label for destination of indirect calls
  - Make sure that every indirect call lands on function entry
- Label for destination of rets and indirect jumps
  - Make sure every indirect jump lands at start of a basic block



Fine-grained CFI (Abadi et al.)

- Statically compute CFG
- Dynamically ensure program never deviates
  - Assign label to each target of indirect transfer
  - Instrument indirect transfers to com
  label of destination with
    the expected label to ensure it's valid



Control Flow Integrity Limitations

Overhead

- Runtime: every indirect branch instruction
- Size: code before indirect branch, encode label at destination •

Scope

- CFI does not protect against data-only attacks
- Needs reliable W^X

How can you defeat CFI?

- Imprecision can allow for control-flow hijacking
  - Can jump to functions that have same label
- Coarse-grained CFI can return to many sites
  - Can use a shadow stack to implement fully precise CFI

oday

- Return-oriented programming
- Control flow integrity

→ Heap corruption

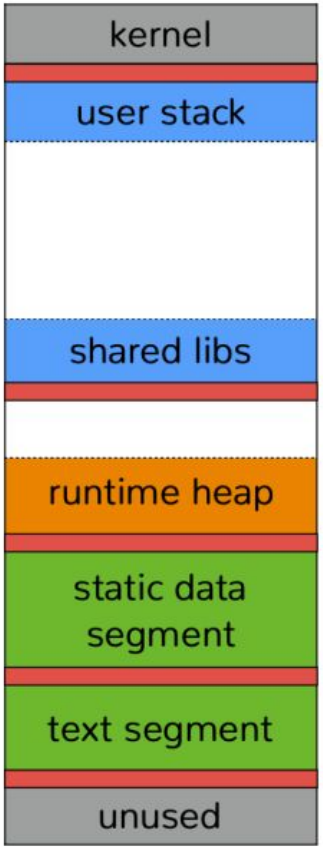
- Isolation

Memory management in C/C++

- C uses explicit memory management
  - Data is allocated and freed dynamically
  - Dynamic memory is accessed via pointers
- You are on your own
  - System does not track memory liveness
  - System doesn’t ensure that pointers are live or valid
- By default C++ has same issues

The heap

- Dynamically allocated data stored on the “heap”
- Heap manager exposes API for allocating and deallocating memory
  - malloc() and free()
  - API invariant: All memory allocated by malloc() has to be released by corresponding call to free()



Heap management

- Organized in contiguous chunks of memory
  - Basic unit of memory
  - Can be free or in use
  - Metadata: size + flags
  - Allocated chunk: payload
- Heap layout evolves with malloc()s and free()s
  - Chunks may get allocated, freed, split, coalesced
- Free chunks are stored in doubly linked lists (bins)
  - Different kinds of bins: fast, unsorted, small, large, ...

How can things go wrong?

- Forget to free memory
- Write/read memory we shouldn’t have access to: Overflow code pointers on the heap
- Use after free: Use pointers that point to freed object
- Double free: Free already freed objects

Most important: heap corruption

- Can bypass security checks (data-only attacks)
  - e.g. isAuthenticated, buffer_size, isAdmin, etc.
- Can overwrite function pointers
  - Direct transfer of control when function is called
  - C++ virtual tables are especially good targets
- Can overwrite heap management data
  - Corrupt metadata in free chunks
  - Program the heap weird machine

Use-after-free in C++

Victim: Free object: `free(obj);`

Attacker: Overwrite the vtable of the object so entry (`obj->vtable[0]`) points to attacker gadget

Victim: Use dangling pointer: `obj->foo()`

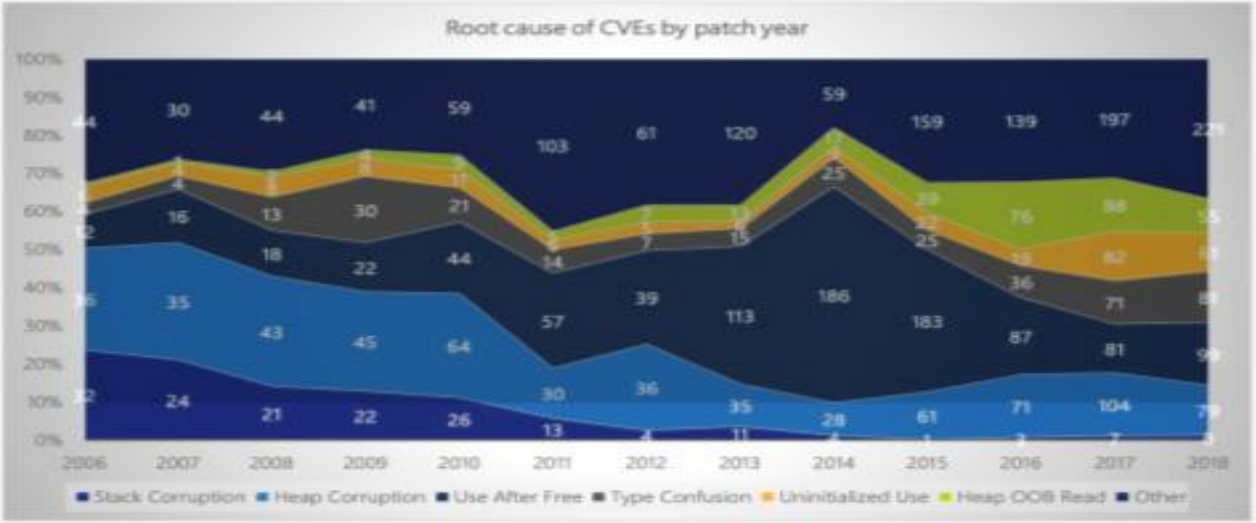
Microsoft

Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape

Matt Miller (@epakskape)
Microsoft Security Response Center (MSRC)

BlueHat IL
February 7th, 2019

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

- #1: heap out-of-bounds
- #2: use after free
- #3: type confusion
- #4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

Heap exploitation mitigations

- Safe heap implementations
  - Safe unlinking
  - Cookies/canaries on the heap
  - Heap integrity check on malloc and free
- Use Rust or a safe garbage collected language

What does all this tell us?

If you’re trying to build a secure system,
use a memory and type-safe language.

Next time on CSE 127…