CSE 127: Introduction to Security

Memory safety and Isolation

George Obaido

UCSD

Spring 2022 Lecture 5

Some slides from Kirill Levchenko, Stefan Savage, Nadia Heninger, Stephen Checkoway, Hovav Shacham, David Wagner, Deian Stefan, Dan Boneh, and Zakir Durumeric

Buffer overflow mitigations

- Avoid unsafe functions: Strcmp, strcpy, gets, etc

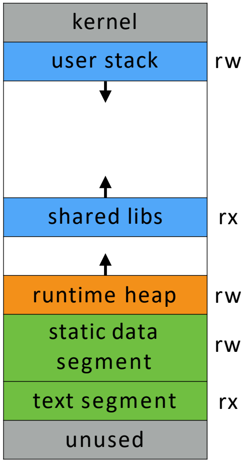
→ Memory writable or executable, not both (W^X)

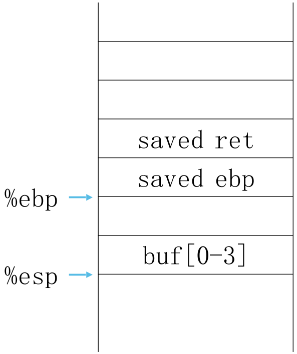
- Address space layout randomization

W^X: write XOR execute

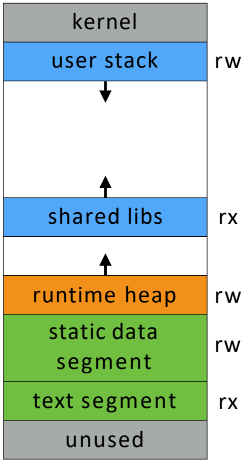
- **Goal:** Prevent execution of shell code from the stack
- **Insight:** Use memory page permission bits
  - Use MMU to ensure memory cannot be both writeable and executable at the same time
- Many names for same idea:
  - XN: eXecute Never
  - W^X: Write XOR eXecute
  - DEP: Data Execution Prevention

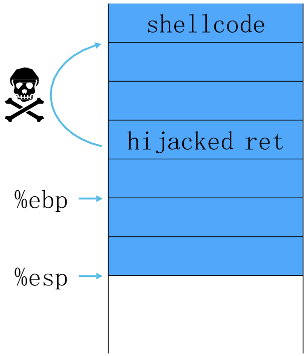
Recall our memory layout





Recall our memory layout



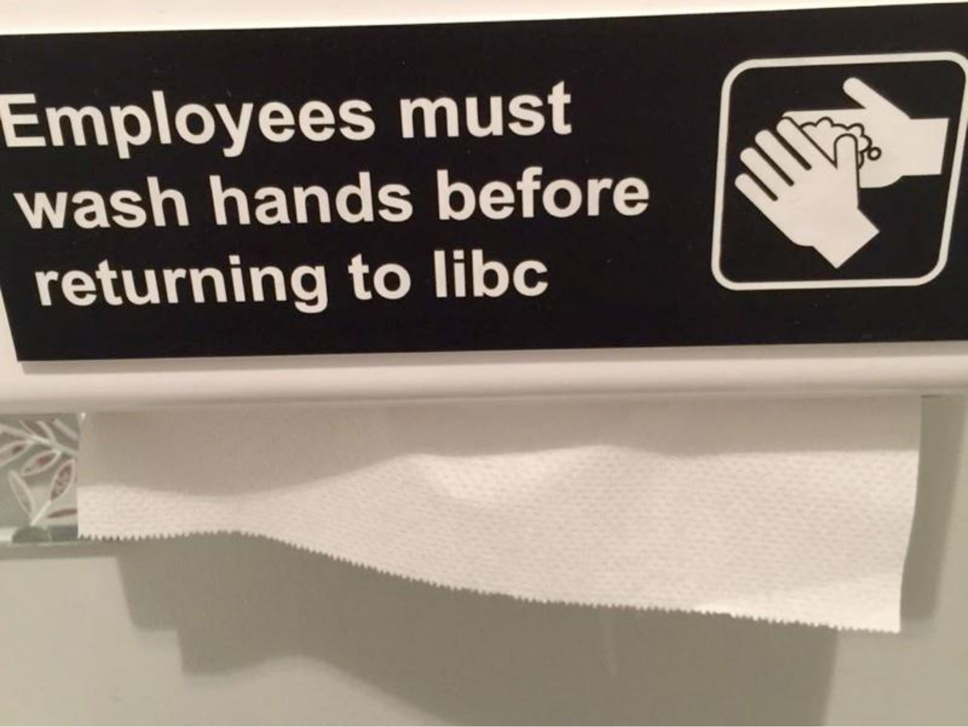


W^X tradeoffs

- **Easy to deploy:** No code changes or recompilation
- **Fast:** Enforced in hardware
  - Downside: What do you do on embedded devices?
- Some pages need to be both writeable and executable
  - Why?

How can we defeat W^X?

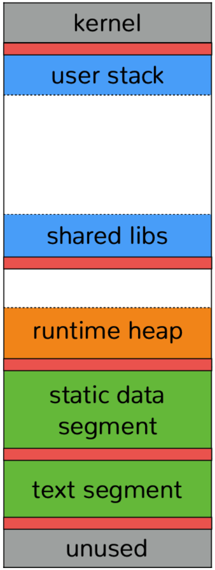
- Can still write to return address stored on the stack
    - Jump to existing code
- Search executable for code that does what you want
    - E.g. if program calls $\text{system}(\text{“/bin/sh”})$ you’re done
    - libc is a good source of code (return-into-libc attacks)


Employees must
wash hands before
returning to libc

Address Space Layout Randomization (ASLR)

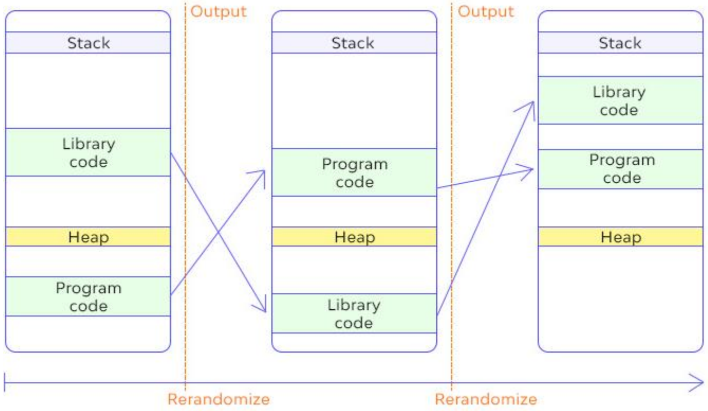
- Traditional exploits need precise addresses
  - **stack-based overflows:** shellcode
  - **return-into-libc:** library addresses

- **Insight:** Make it harder for attacker to guess location of shellcode/libc by randomizing the address of different memory regions



How much do we randomize?

32-bit PaX ASLR (x86)



ASLR Tradeoffs

- **Intrusive:** Need compiler, liker, loader support
    - Process layout must be randomized
    - Programs must be compiled to not have absolute jumps
- **Incurs overhead:** Increases code size and performance overhead
- Also mitigates heap-based overflow attacks

When do we randomize?

Many options.

- At boot?
- At compile/link time?
- At run/load time?

What’s the tradeoff?

- Not useful for forensic analysis

How can we defeat ASLR?

- **-fno-pie** binaries have fixed code and data addresses
    - Enough to carry out control flow hijacking attacks

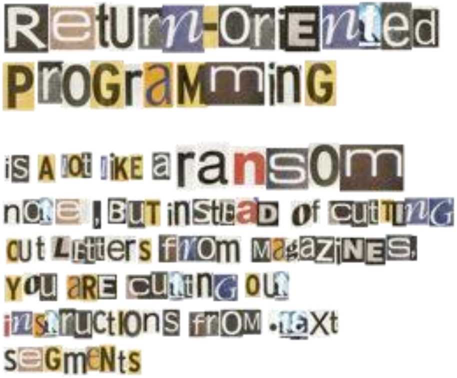
- Each region has random offset, but layout is fixed
    - Single address in a region leaks every address in region

Today

- Return-oriented programming
- Heap corruption
- Isolation

Return-Oriented Programming (ROP)

- 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

is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from text segments

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 assembly instructions?
  - End of function **(inserted by compiler)**

One ret, multiple gadgets

<u>b8 01 00 00 00 5b c9 c3</u> =

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

One ret, multiple gadgets

b8 01 $\underline{\text{00 00 00 5b c9 c3}}$ =

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

One ret, multiple gadgets

b8 01 00 00 $\underline{\texttt{00 5b c9 c3}}$ =

```
add %bl,-0x37(%eax)
ret
```

One ret, multiple gadgets

b8 01 00 00 00 $\underline{\text{5b c9 c3}}$ =

```
pop %ebx
leave
ret
```

One ret, multiple gadgets

b8 01 00 00 00 5b <u>c9 c3</u> = leave
ret

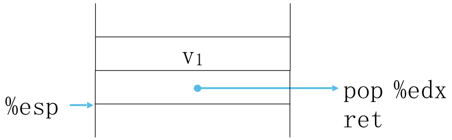
One ret, multiple gadgets

b8 01 00 00 00 5b c9 <u>c3</u> = 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?

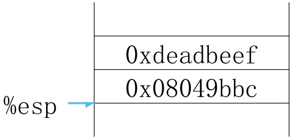
What happens if this is what we overflow the stack with?



relevant register(s):

```
%edx = 0x00000000
```

relevant stack:



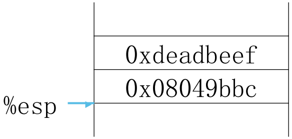
relevant code:

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

relevant register(s):

```
%edx = 0x00000000
```

relevant stack:



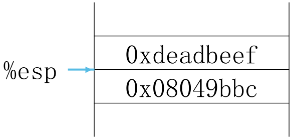
relevant code:

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

relevant register(s):

```
%edx = 0x00000000
```

relevant stack:



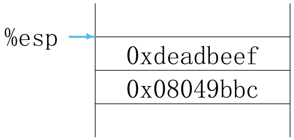
relevant code:

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

relevant register(s):

```
%edx = 0xdeadbeef
```

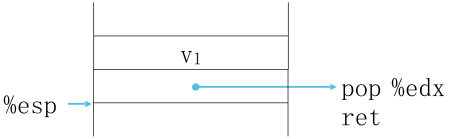
relevant stack:



relevant code:

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

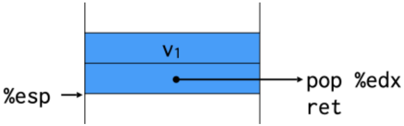
This is a ROP gadget!



movl v_1, %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



Can express arbitrary programs

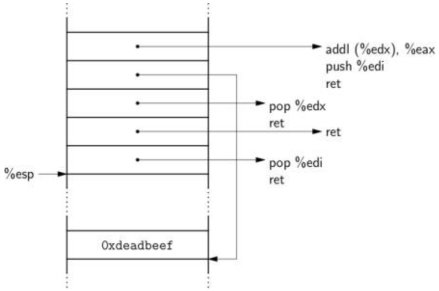


Figure 5: Simple add into %eax.

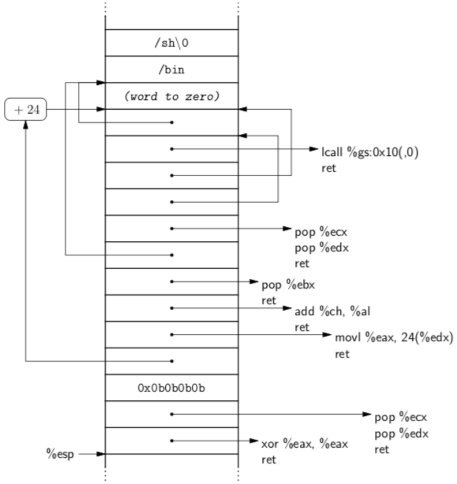
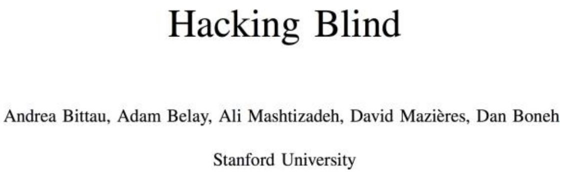


Figure 16: Shellcode.

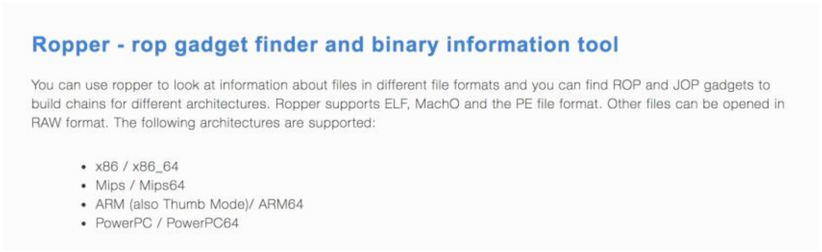
Can find gadgets automatically



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).

- One ROP mitigation could be a Address Space Layout Randomization **(ASLR)**

Today

- Return-oriented programming

→ Heap corruption

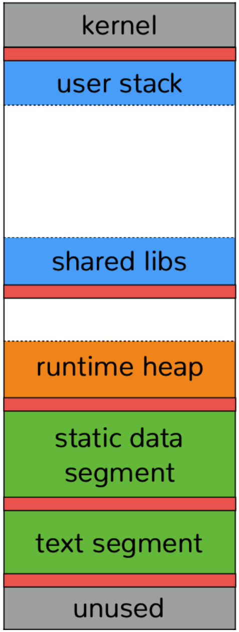
- Isolation

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
    - If system does not track memory liveness
    - If 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
- Heap layout evolves with malloc()s and free()s
    - Chunks may get allocated and freed
- 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 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()

Dangling pointers and memory leaks

- Dangling pointer: Pointer points to a memory location that no longer exists
- Memory leaks (tardy free) Memory in heap that can no longer be accessed

Dangling pointer

```
int main(){
 int *arr1 = malloc(sizeof(int));
  *arr1 = 2;
  printf("%d/n", *arr1)
  free(arr1);
  arr1 = NULL //Solution: Set to Null
  return 0;
}
```

Memory Leak

```
int main(int argc, char *arg[]){
  int *arr1 = malloc(sizeof(int));
  *arr1 = 2;
  printf("%d/n", *arr1)
  free(arr1);//solution: free the memory or
deallocate the memory
  arr1 = NULL
  return 0;
}
```

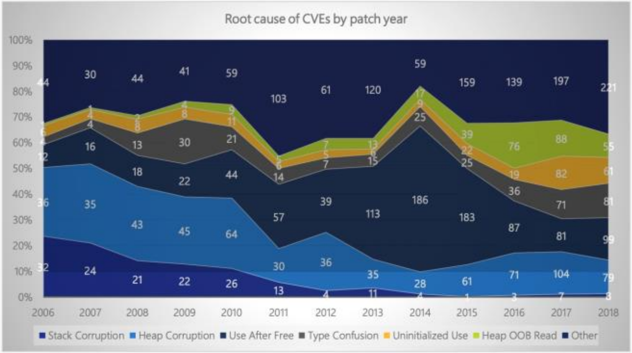
Microsoft

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, such as Julia, Ruby, etc.

What does all this tell us?

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

Today

- Understand basic principles for building secure systems
- Understand mechanisms used to build secure systems

Running untrusted code

We often need to run buggy or untrusted code.

Running untrusted code

We often need to run buggy or untrusted code.

- Desktop applications
- Mobile apps
- Untrusted user code
- Web sites, Javascript, browser extensions
- PDF viewers, email clients
- VMs on cloud computing infrastructure

Systems must be designed to be resilient in the face of vulnerabilities and malicious users.

Principles of secure system design

- Least privilege
- Privilege separation
- Complete mediation
- Fail safe/closed
- Defense in depth
- Keep it simple