Mitigating Memory Safety Vulnerabilities

CS 161 Spring 2024 - Lecture 5

Next: Memory Safety Mitigations

- Memory-safe languages
- Writing memory-safe code
- Building secure software
- Exploit mitigations
    - Non-executable pages
    - Stack canaries
    - Pointer authentication
    - Address space layout randomization (ASLR)
- Combining mitigations

Today: Defending Against Memory Safety Vulnerabilities

- We’ve seen how widespread and dangerous memory safety vulnerabilities can be. Why do these vulnerabilities exist?
    - Programming languages aren’t designed well for security.
    - Programmers often aren’t security-aware.
    - Programmers write code without designing security in from the start.
    - Programmers are humans. Humans make mistakes.

Today: Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
    - Use safer programming languages.
    - Learn to write memory-safe code.
    - Use tools for analyzing and patching insecure code.
    - Add mitigations that make it harder to exploit common vulnerabilities.

Using Memory-Safe Languages

Textbook Chapter 4.1

Today: Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
  - Use safer programming languages.
  - Learn to write memory-safe code.
  - Use tools for analyzing and patching insecure code.
  - Add mitigations that make it harder to exploit common vulnerabilities.

Memory-Safe Languages

- **Memory-safe languages** are designed to check bounds and prevent undefined memory accesses
- By design, memory-safe languages are not vulnerable to memory safety vulnerabilities
    - Using a memory-safe language is the **only** way to stop 100% of memory safety vulnerabilities
- Examples: Java, Python, C#, Go, Rust
    - Most languages besides C, C++, and Objective C

Why Use Non-Memory-Safe Languages?

- Most commonly-cited reason: **performance**
- Comparison of memory allocation performance
  - C and C++ (not memory safe): **`malloc`** usually runs in (amortized) constant-time
  - Java (memory safe): The garbage collector may need to run at any arbitrary point in time, adding a 10–100 ms delay as it cleans up memory

The Cited Reason: The Myth of Performance

- For most applications, the performance difference from using a memory-safe language is insignificant
  - Possible exceptions: Operating systems, high performance games, some embedded systems
- C’s improved performance is not a direct result of its security issues
  - Historically, safer languages were slower, so there was a tradeoff
  - Today, safe alternatives have comparable performance (e.g. Go and Rust)
  - Secure C code (with bounds checking) ends up running as quickly as code in a memory-safe language anyway
  - You don’t need to pick between security and performance: You can have both!

The Cited Reason: The Myth of Performance

- Programmer time matters too
    - You save more time writing code in a memory-safe language than you save in performance
- “Slower” memory-safe languages often have libraries that plug into fast, secure, C libraries anyway
    - Example: NumPy in Python (memory-safe)

The Real Reason: Legacy

- Most common actual reason: inertia and **legacy**
- Huge existing code bases are written in C, and building on existing code is easier than starting from scratch
    - If old code is written in {language}, new code will be written in {language}!

Example of Legacy Code: iPhones

- When Apple created the iPhone, they modified their existing OS and environment to run on a phone
- Although there may be very little code dating back to 1989 on your iPhone, many of the programming concepts remained!
- If you want to write apps on an iPhone, you still often use Objective C
- **Takeaway**: Non-memory-safe languages are still used for legacy reasons

Writing Memory-Safe Code

Textbook Chapter 4.2

Today: Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
    - Use safer programming languages.
    - Learn to write memory-safe code.
    - Use tools for analyzing and patching insecure code.
    - Add mitigations that make it harder to exploit common vulnerabilities.

Writing Memory-Safe Code

- Defensive programming: Always add checks in your code just in case
  - Example: Always check a pointer is not null before dereferencing it, even if you’re sure the pointer is going to be valid
  - Relies on programmer discipline
- Use safe libraries
  - Use functions that check bounds
  - Example: Use **`fgets`** instead of **`gets`**
  - Example: Use **`strncpy`** or **`strlcpy`** instead of **`strcpy`**
  - Example: Use **`snprintf`** instead of **`sprintf`**
  - Relies on programmer discipline or tools that check your program

Writing Memory-Safe Code

- Structure user input
  - Constrain how untrusted sources can interact with the system
  - Example: When asking a user to input their age, only allow digits (0–9) as inputs
- Reason carefully about your code
  - When writing code, define a set of *preconditions*, *postconditions*, and *invariants* that must be satisfied for the code to be memory-safe
  - Very tedious and rarely used in practice, so it’s out of scope for this class

Building Secure Software

Textbook Chapter 4.3

Today: Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
  - Use safer programming languages.
  - Learn to write memory-safe code.
  - Use tools for analyzing and patching insecure code.
  - Add mitigations that make it harder to exploit common vulnerabilities.

Approaches for Building Secure Software/Systems

- Run-time checks
    - Automatic bounds-checking
    - May involve performance overhead
    - Crash if the check fails
- Monitor code for run-time misbehavior
    - Example: Look for illegal calling sequences
    - Example: Your code never calls **`execve`**, but you notice that your code is executing **`execve`**
    - Probably too late by the time you detect it
- Contain potential damage
    - Example: Run system components in sandboxes or virtual machines (VMs)
    - Think about privilege separation

Approaches for Building Secure Software/Systems

- Bug-finding tools
  - Excellent resource, as long as there aren’t too many false bugs
- Code review
  - Hiring someone to look over your code for memory safety errors
  - Can be very effective… but also expensive
- Vulnerability scanning
  - Probe your systems for known flaws
- Penetration testing (“pen-testing”)
  - Pay someone to break into your system

Testing for Software Security Issues

- How can we test programs for memory safety vulnerabilities?
  - Fuzz testing: Random inputs
  - Use tools like Valgrind (tool for detecting memory leaks)
  - Test corner cases
- How do we tell if we've found a problem?
  - Look for a crash or other unexpected behavior
- How do we know that we've tested enough?
  - Hard to know, but code-coverage tools can help

Working Towards Secure Systems

- Modern software often imports lots of different libraries
  - Libraries are often updated with security patches
  - It’s not enough to keep your own code secure: You also need to keep libraries updated with the latest security patches!
- What’s hard about patching?
  - Can require restarting production systems
  - Can break crucial functionality

Exploit Mitigations

Textbook Chapter 4.4

Today: Defending Against Memory Safety Vulnerabilities

- What are some approaches to defending against memory safety vulnerabilities?
    - Use safer programming languages.
    - Learn to write memory-safe code.
    - Use tools for analyzing and patching insecure code.
    - Add mitigations that make it harder to exploit common vulnerabilities.

Exploit Mitigations

- Scenario
    - Someone has just handed you a large, existing codebase
    - It’s not written in a memory-safe language, and it wasn’t written with memory safety in mind
    - How can you protect this code from exploits without having to completely rewrite it?
- **Exploit mitigations (code hardening)**: Compiler and runtime defenses that make common exploits harder
    - Find ways to turn attempted exploits into program crashes
    - Crashing is safer than exploitation: The attacker can crash our system, but at least they can’t execute arbitrary code
    - Mitigations are cheap (low overhead) but not free (some costs associated with them)

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
4. Return from the function
5. Begin executing malicious shellcode

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
4. Return from the function
5. Begin executing malicious shellcode

We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Mitigation: Non-Executable Pages

Textbook Chapter 4.5 & 4.6 & 4.7

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
4. Return from the function
5. Begin executing malicious shellcode
    - Mitigation: Non-executable pages

We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Non-Executable Pages

- Idea: Most programs don't need memory that is both written to and executed, so make portions of memory **either** executable **or** writable but not both
  - Stack, heap, and static data: Writable but not executable
  - Code: Executable but not writable
- Page table entries have a writable bit and an executable bit that can be set to achieve this behavior
  - Recall page tables from 61C: Converts virtual addresses to physical addresses
  - Implemented in hardware, so effectively 0 overhead!
- Also known as
  - **W^X** (write XOR execute)
  - **DEP** (Data Execution Prevention, name used by Windows)
  - **No-execute bit** (the name of the bit itself)

Subverting Non-Executable Pages

- Issue: Non-executable pages doesn’t prevent an attacker from leveraging **existing** code in memory as part of the exploit
- Most programs have many functions loaded into memory that can be used for malicious behavior
    - **Return-to-libc**: An exploit technique that overwrites the RIP to jump to a functions in the standard C library (libc) or a common operating system function
    - **Return-oriented programming (ROP)**: Constructing custom shellcode using pieces of code that already exist in memory

Subverting Non-Executable Pages: Return-to-libc

- Recall: Per the x86 calling convention, each program expects arguments to be placed directly above the RIP
- Consider the **`system`** function, which executes a shell command. We want to execute it like this:

```
char cmd[] = "rm -rf /";
system(cmd);
```

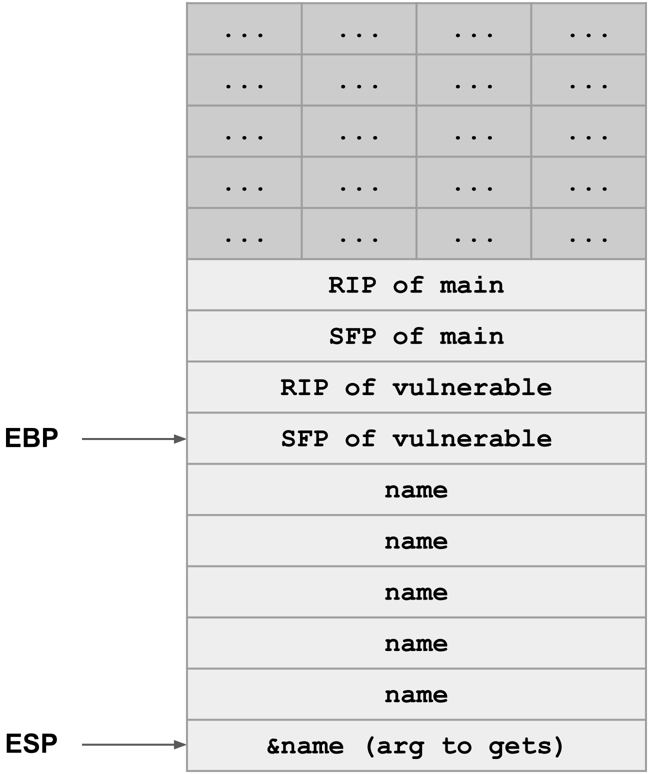
Subverting Non-Executable Pages: Return-to-libc

Exploit:

```
'A' * 24
    + [address of system]
    + 'B' * 4
    + [address of "rm -rf /"]
    + "rm -rf /"
```

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

```
system:
    ...

vulnerable:
    ...
EIP →   call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...

RIP of main

SFP of main

RIP of vulnerable

EBP → SFP of vulnerable

name

name

name

name

name

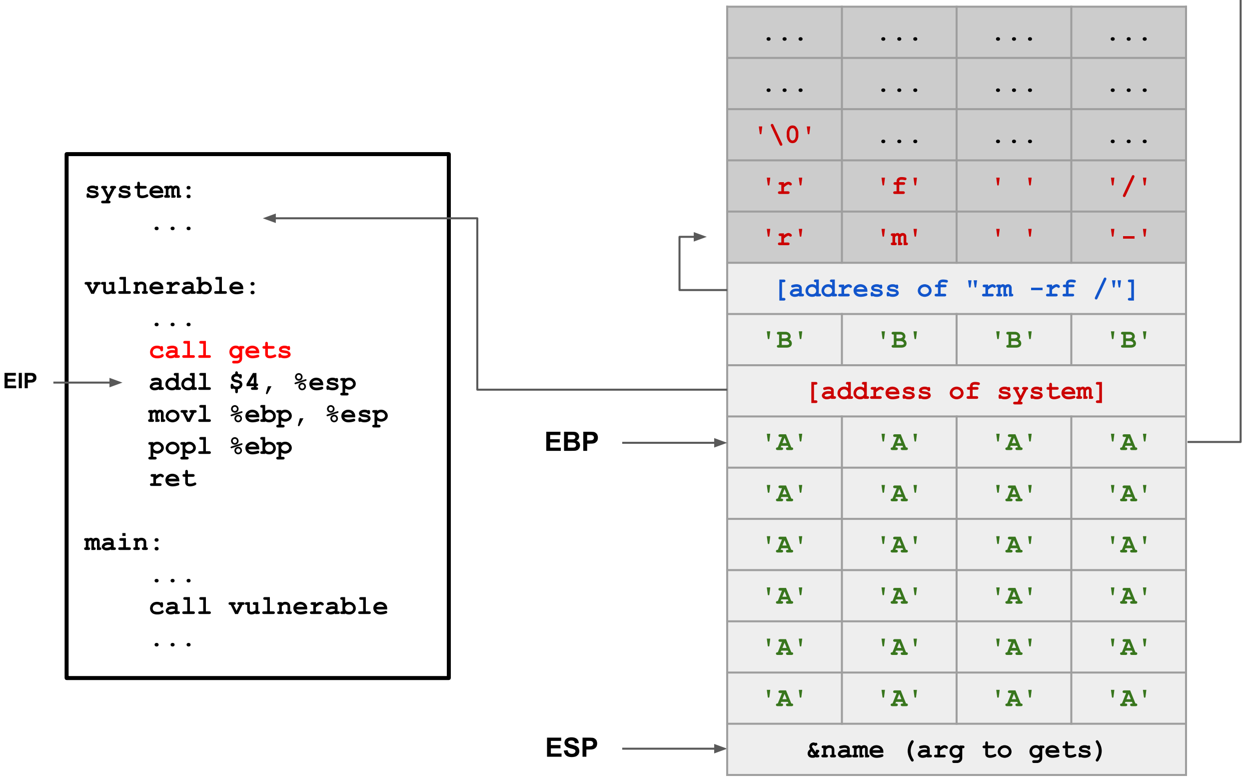
ESP → &name (arg to gets)

Subverting Non-Executable Pages: Return-to-libc

Exploit:

```
'A' * 24
    + [address of system]
    + 'B' * 4
    + [address of "rm -rf /"]
    + "rm -rf /"
```

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

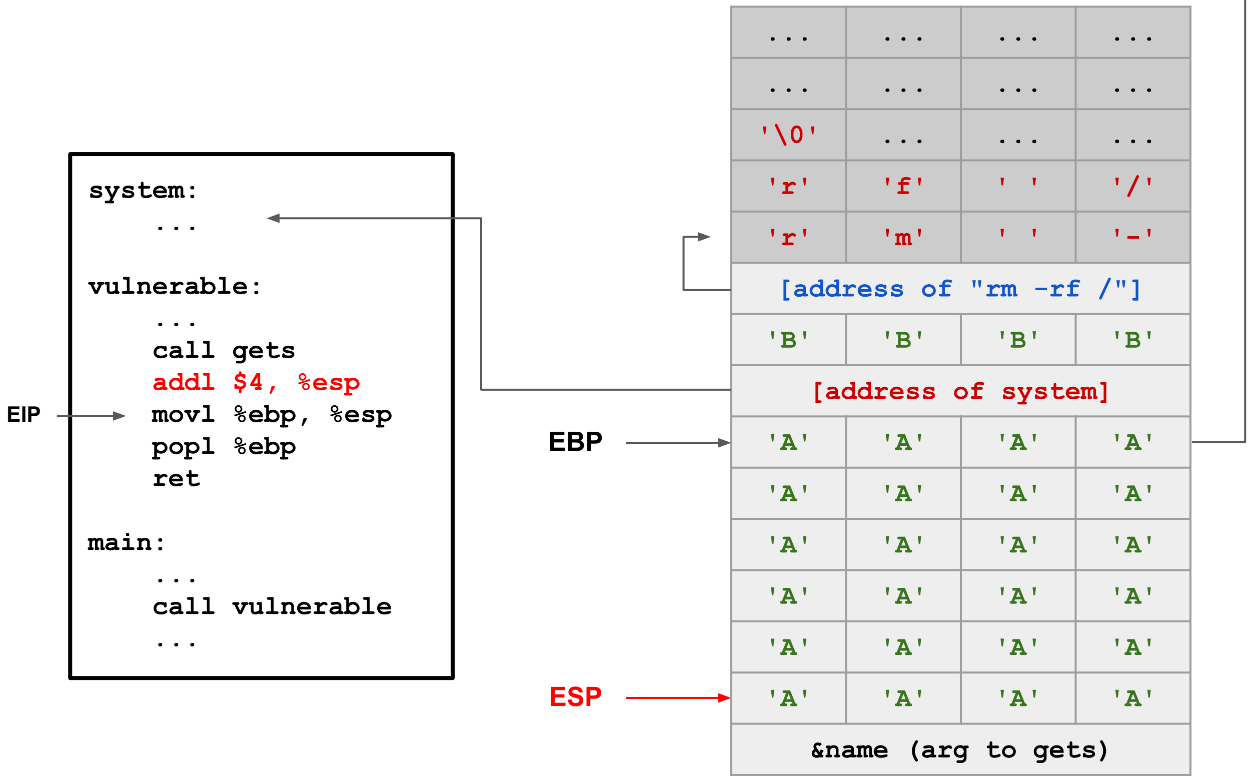


```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

...	...	...	...
...	...	...	...
'\0'	...	...	...
'r'	'f'	' '	'/'
'r'	'm'	' '	'-'
[address of "rm -rf /"]			
'B'	'B'	'B'	'B'
[address of system]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

Subverting Non-Executable Pages: Return-to-libc

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

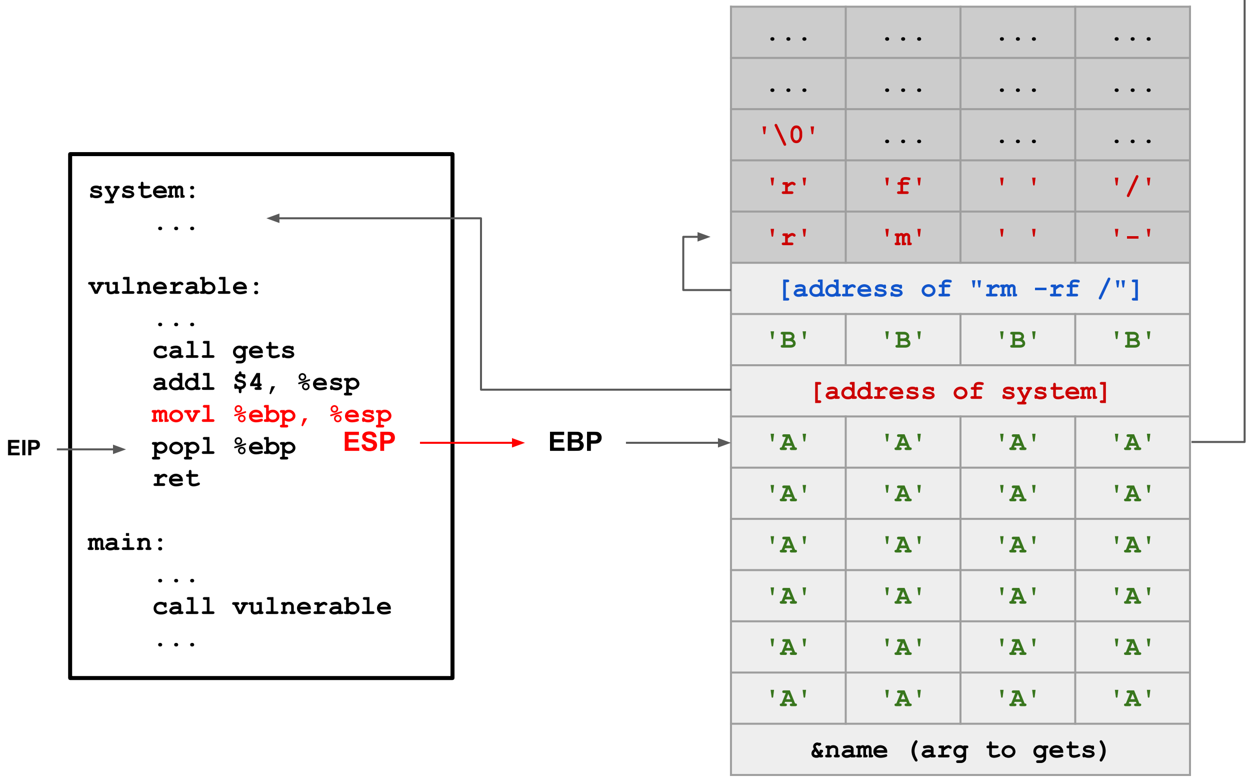


```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

...	...	...	...
...	...	...	...
'\0'	...	...	...
'r'	'f'	' '	'/'
'r'	'm'	' '	'-'
[address of "rm -rf /"]			
'B'	'B'	'B'	'B'
[address of system]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

Subverting Non-Executable Pages: Return-to-libc

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

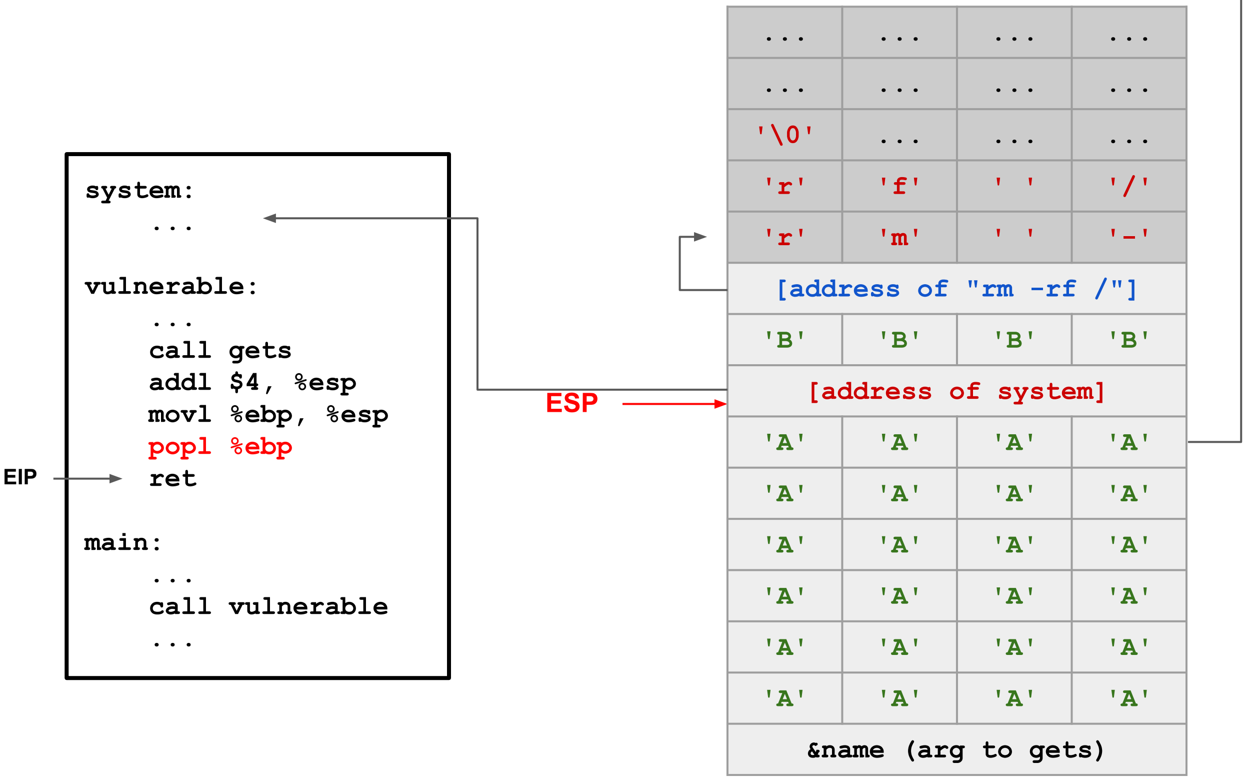


```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

...	...	...	...
...	...	...	...
'\0'	...	...	...
'r'	'f'	' '	'/'
'r'	'm'	' '	'-'
[address of "rm -rf /"]			
'B'	'B'	'B'	'B'
[address of system]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

Subverting Non-Executable Pages: Return-to-libc

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```



```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

...	...	...	...
...	...	...	...
'\0'	...	...	...
'r'	'f'	' '	'/'
'r'	'm'	' '	'-'
[address of "rm -rf /"]			
'B'	'B'	'B'	'B'
[address of system]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

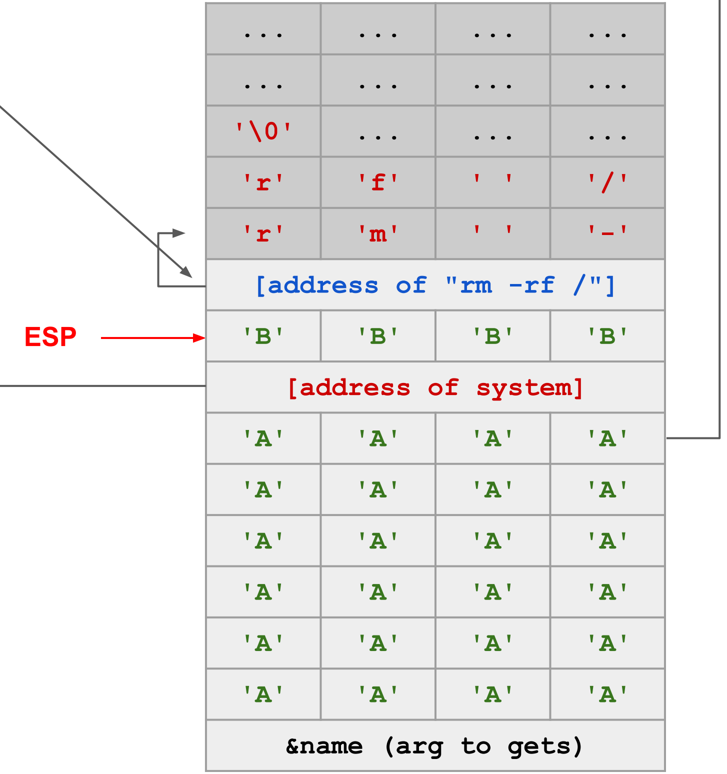
Subverting Non-Executable Pages: Return-to-libc

We jumped into the **`system`** function, and it expects the first argument to be 4 bytes above the ESP: **`"rm -rf /"`**!

```
int system(char *command);

void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP

```
system:
    ...

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



...	...	...	...
...	...	...	...
'\0'	...	...	...
'r'	'f'	' '	'/'
'r'	'm'	' '	'-'
[address of "rm -rf /"]			
'B'	'B'	'B'	'B'
[address of system]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

Subverting Non-Executable Pages: ROP

- Instead of executing an existing function, execute your own code by executing different pieces of different code!
  - We don't need to jump to the beginning of a function: We can jump into the middle of it to just take the code chunks that we need
- **Gadget**: A small set of assembly instructions that already exist in memory
  - Gadgets usually end in a `ret` instruction
  - Gadgets are usually **not** full functions
- ROP strategy: We write a chain of return addresses starting at the RIP to achieve the behavior we want
  - Each return address points to a gadget
  - The gadget executes its instructions and ends with a `ret` instruction
  - The `ret` instruction jumps to the address of the next gadget on the stack

Subverting Non-Executable Pages: ROP

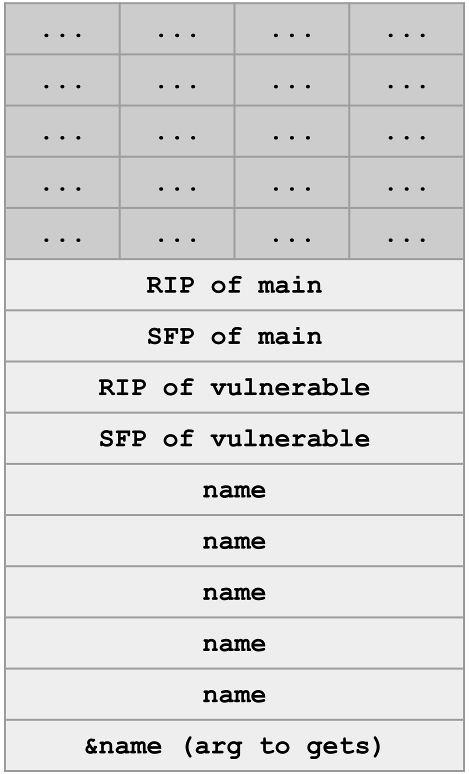
Example: Let’s say our shellcode involves the following sequence:

```
movl $1, %eax
xorl %eax, %ebx
```

The following is present in memory:

```
foo:
          ...
<foo+7>  addl $4, %esp
<foo+10> xorl %eax, %ebx
<foo+12> ret

bar:
          ...
<bar+22> andl $1, %edx
<bar+25> movl $1, %eax
<bar+30> ret
```

How can we chain returns to run the code sequence we want?



...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
RIP of main			
SFP of main			
RIP of vulnerable			
SFP of vulnerable			
name			
name			
name			
name			
name			
&name (arg to gets)			

Subverting Non-Executable Pages: ROP

Example: Let’s say our shellcode involves the following sequence:

```
movl $1, %eax
xorl %eax, %ebx
```

The following is present in memory:

```
foo:
          ...
<foo+7>  addl $4, %esp
<foo+10> xorl %eax, %ebx
<foo+12> ret

bar:
      ...
<bar+22> andl $1, %edx
<bar+25> movl $1, %eax
<bar+30> ret
```

If we jump 25 bytes after the start of **bar** then 10 bytes after the start of **foo**, we get the result we want!

How can we chain returns to run the code sequence we want?

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
RIP of main			
SFP of main			
RIP of vulnerable			
SFP of vulnerable			
name			
name			
name			
name			
name			
&name (arg to gets)			

Subverting Non-Executable Pages: ROP

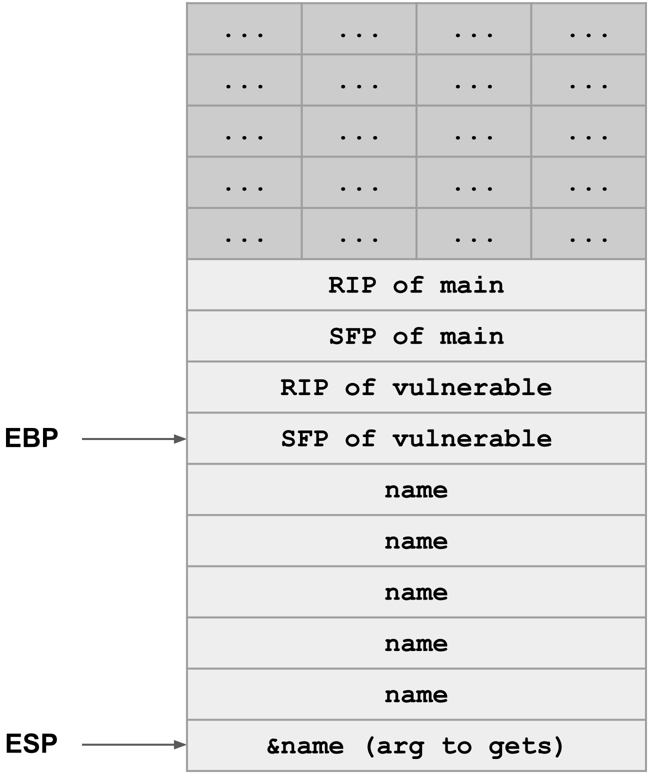
Exploit:

```
'A' * 24
    + [address of <bar+25>]
    + [address of <foo+10>]
    + ... (more chains)
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP

```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



Subverting Non-Executable Pages: ROP

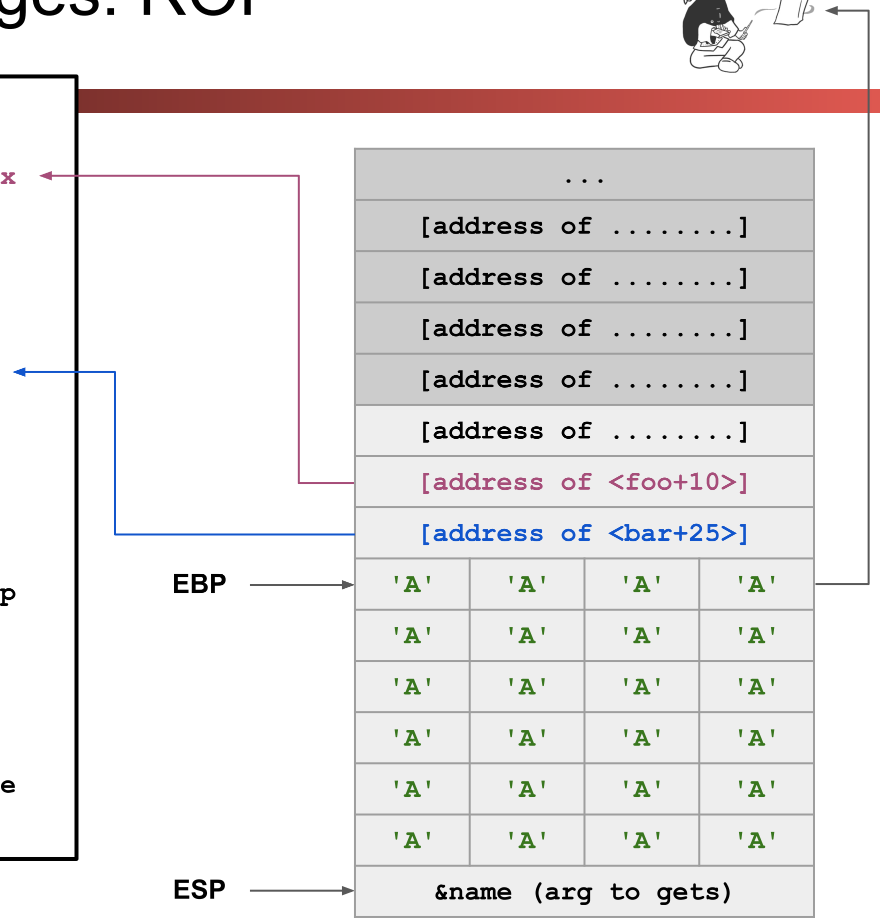
Exploit:

```
'A' * 24
    + [address of <bar+25>]
    + [address of <foo+10>]
    + ... (more chains)
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

EIP

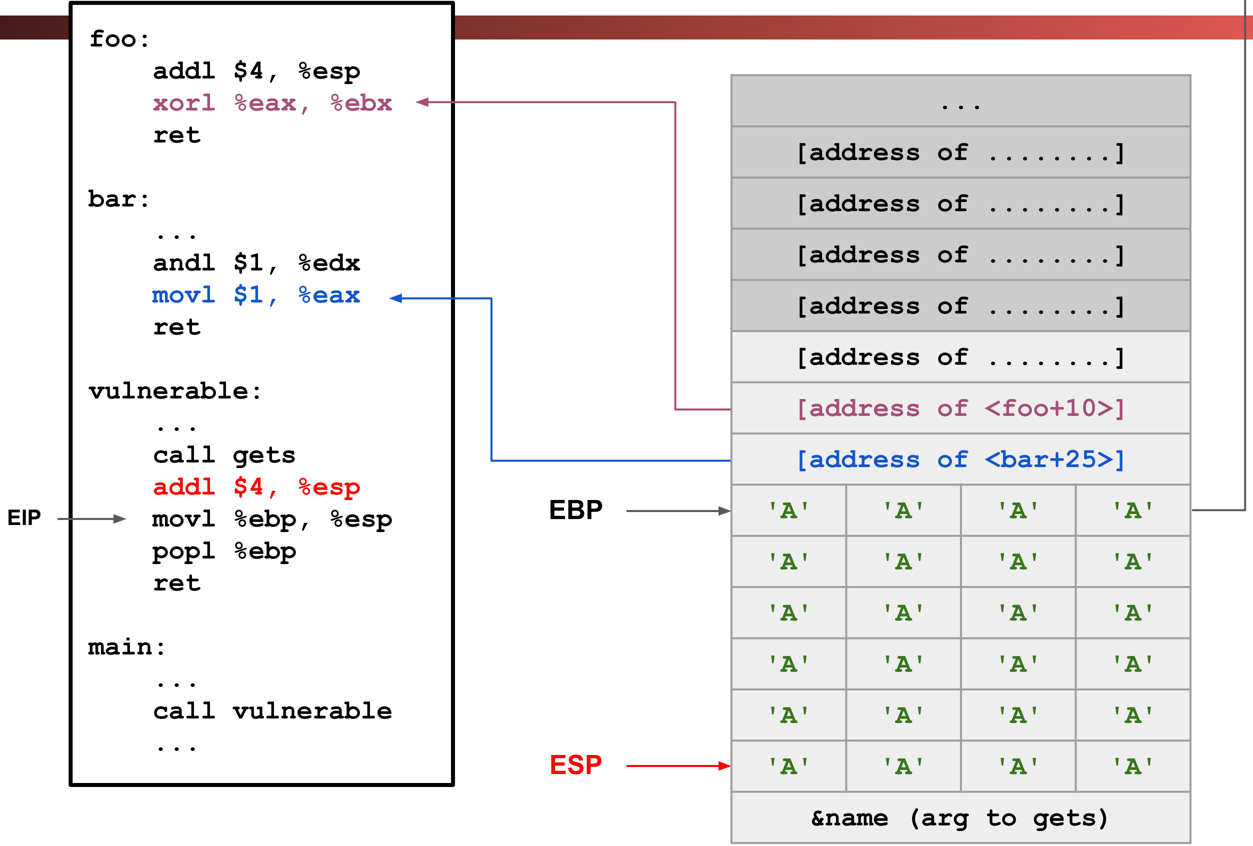
```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```



...			
[address of ....]			
[address of ....]			
[address of ....]			
[address of ....]			
[address of ....]			
[address of <foo+10>]			
[address of <bar+25>]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

Subverting Non-Executable Pages: ROP

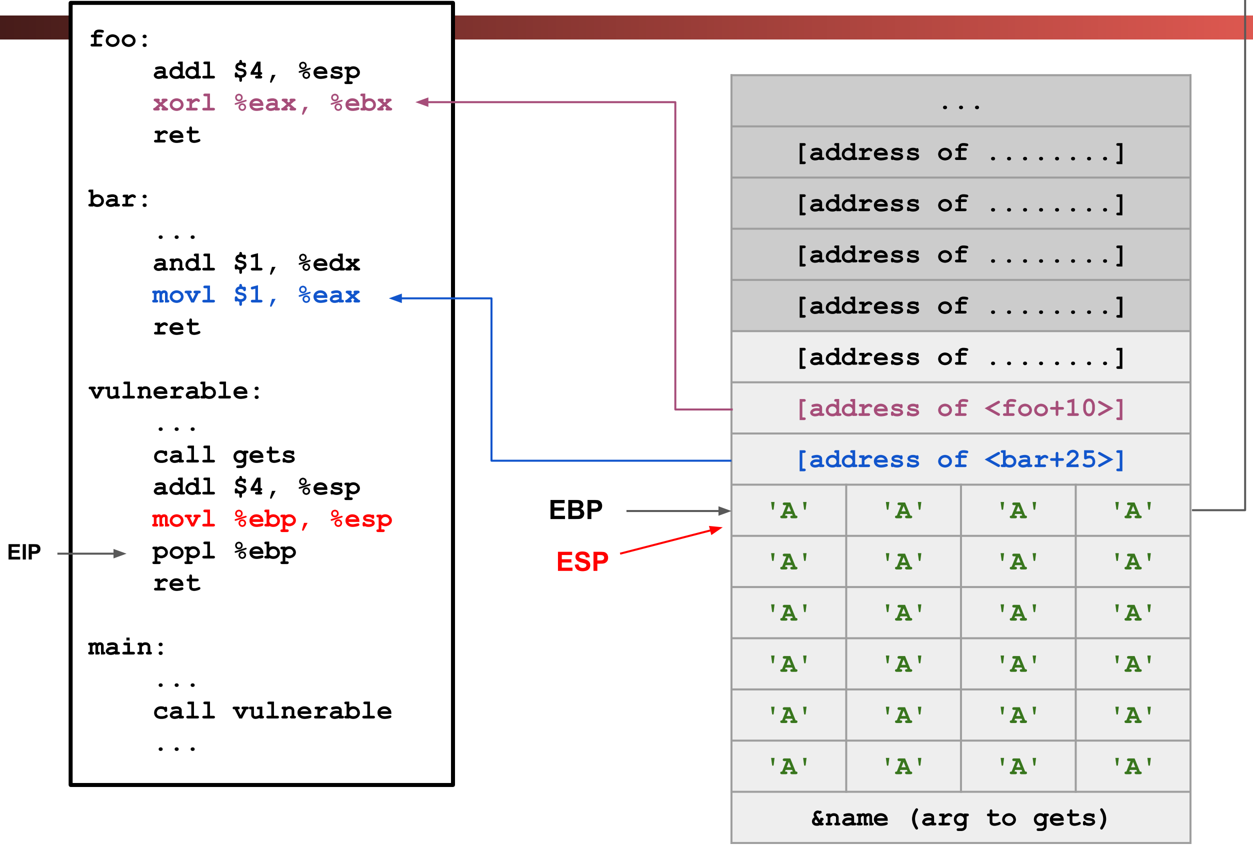
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```



```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

Subverting Non-Executable Pages: ROP

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

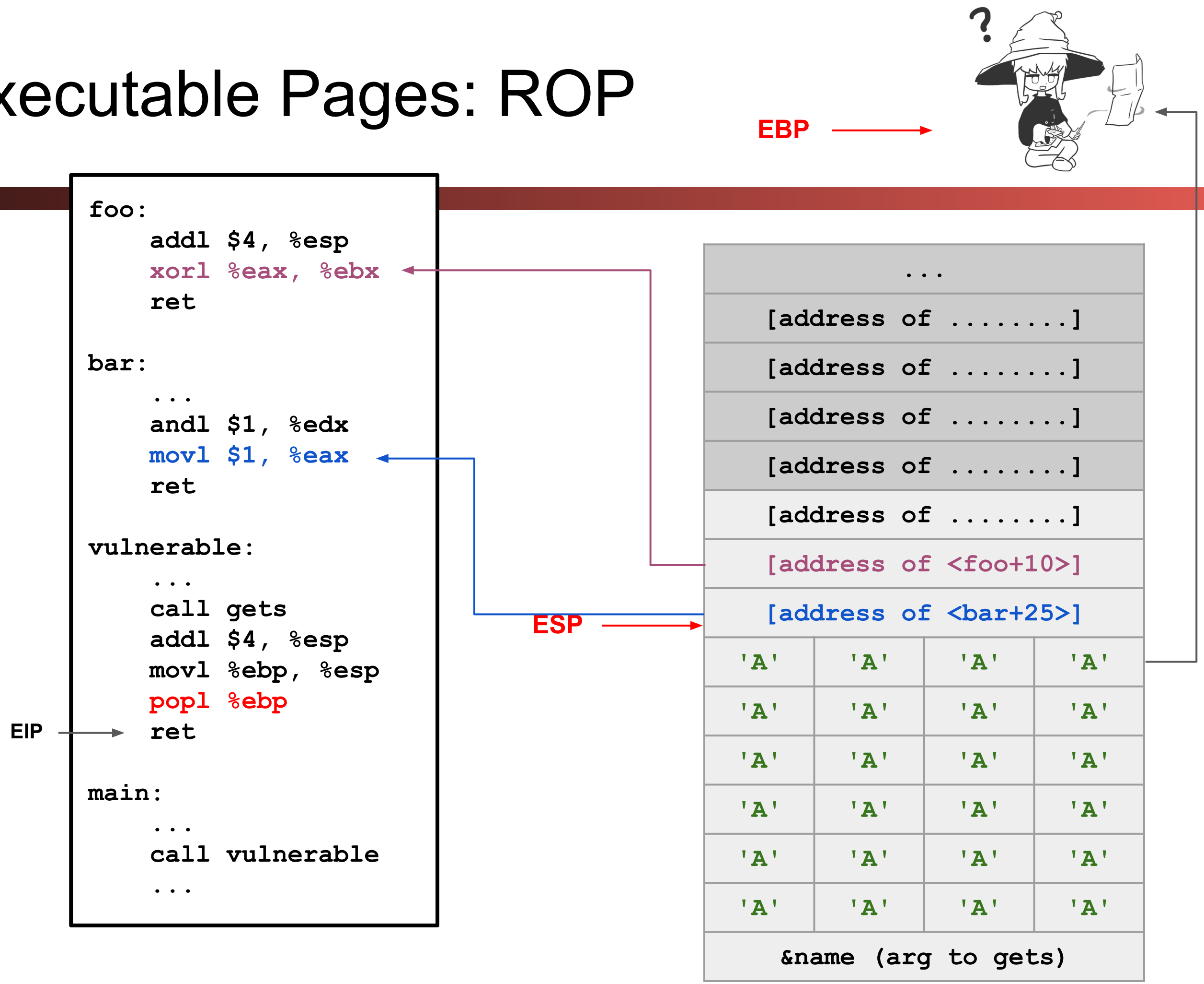


```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

...			
[address of ....]			
[address of ....]			
[address of ....]			
[address of ....]			
[address of ....]			
[address of <foo+10>]			
[address of <bar+25>]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

Subverting Non-Executable Pages: ROP

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

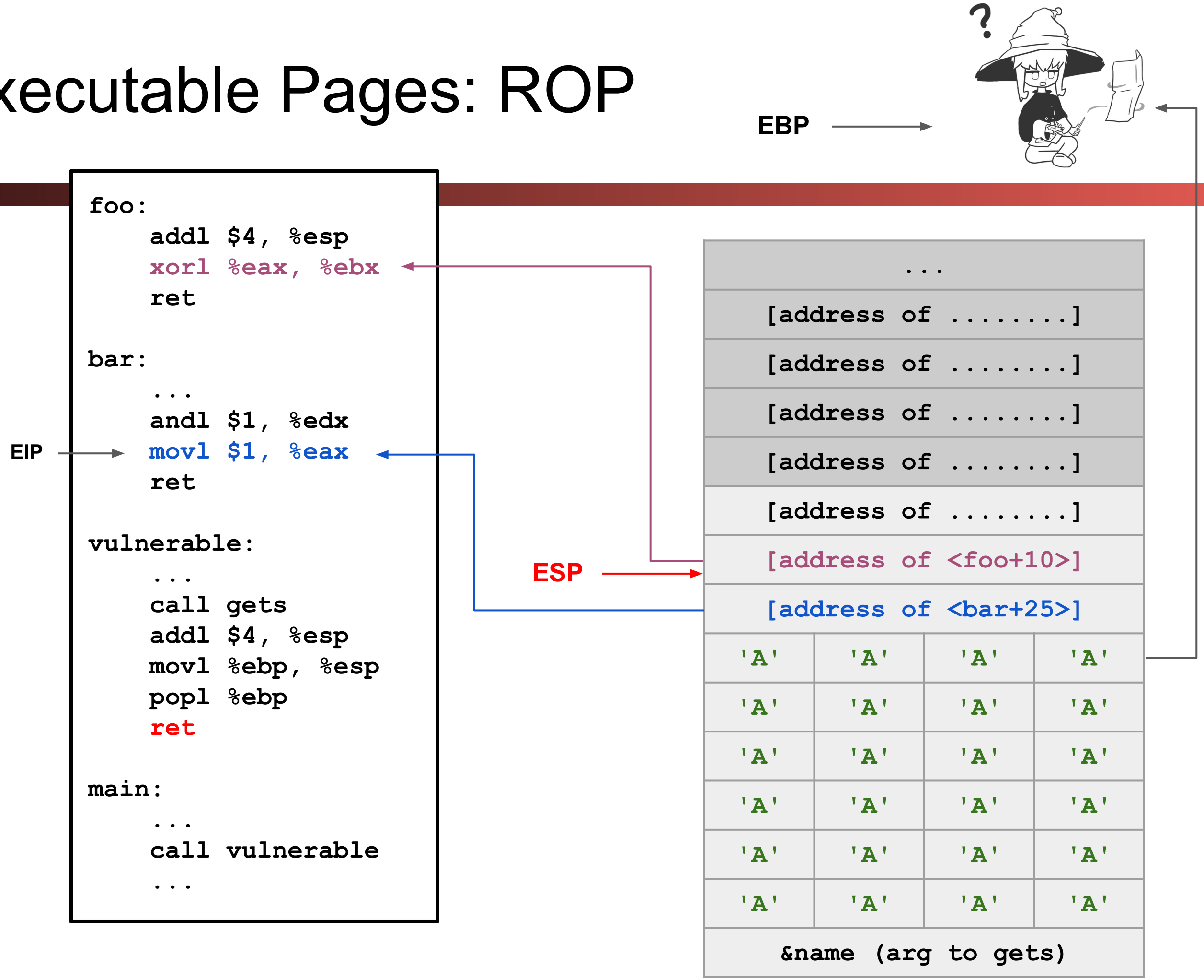


```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

...			
[address of ....]			
[address of ....]			
[address of ....]			
[address of ....]			
[address of ....]			
[address of <foo+10>]			
[address of <bar+25>]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

Subverting Non-Executable Pages: ROP

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

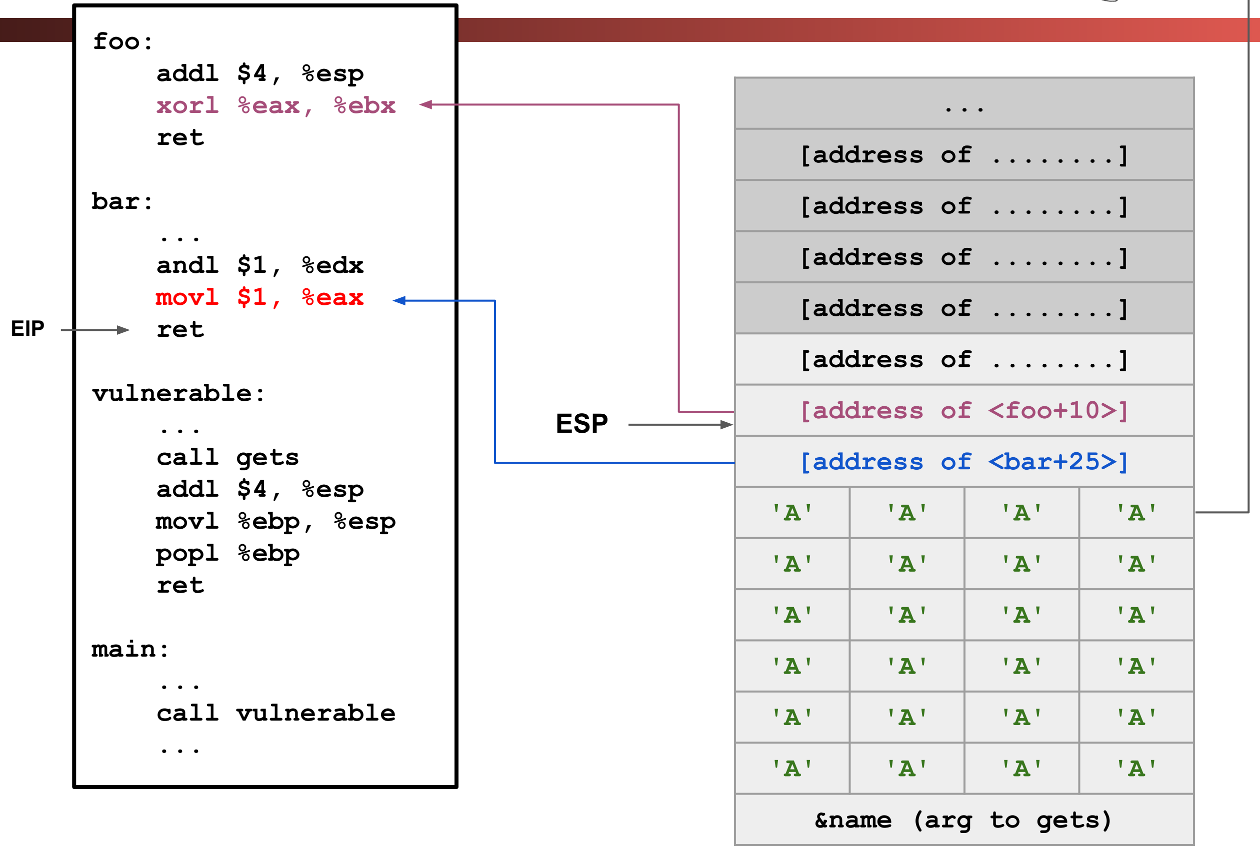


```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

```
...
[address of ........]
[address of ........]
[address of ........]
[address of ........]
[address of ........]
[address of <foo+10>]
[address of <bar+25>]
'A'  'A'  'A'  'A'
'A'  'A'  'A'  'A'
'A'  'A'  'A'  'A'
'A'  'A'  'A'  'A'
'A'  'A'  'A'  'A'
'A'  'A'  'A'  'A'
&name (arg to gets)
```

Subverting Non-Executable Pages: ROP

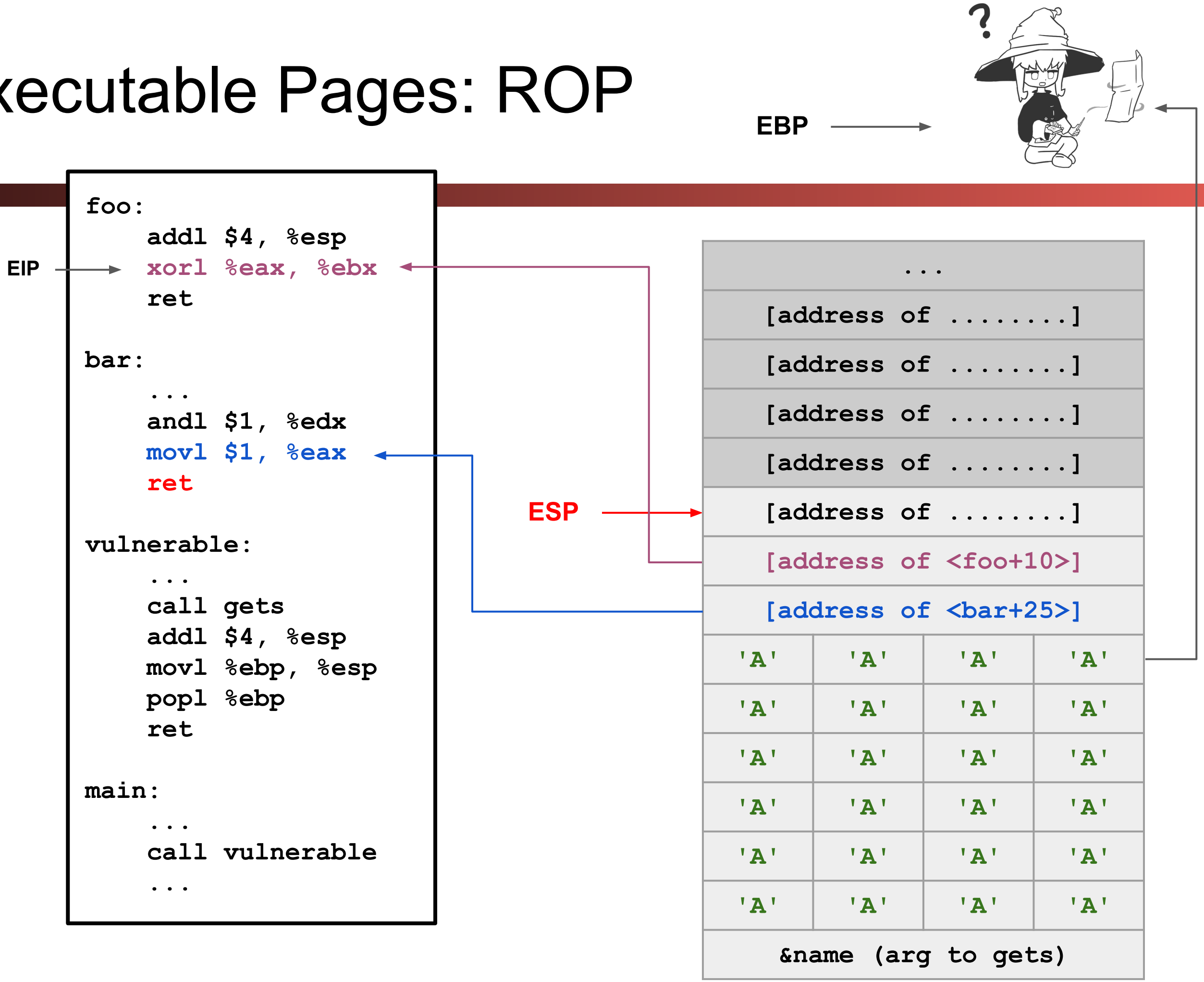
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```



```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

Subverting Non-Executable Pages: ROP

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```



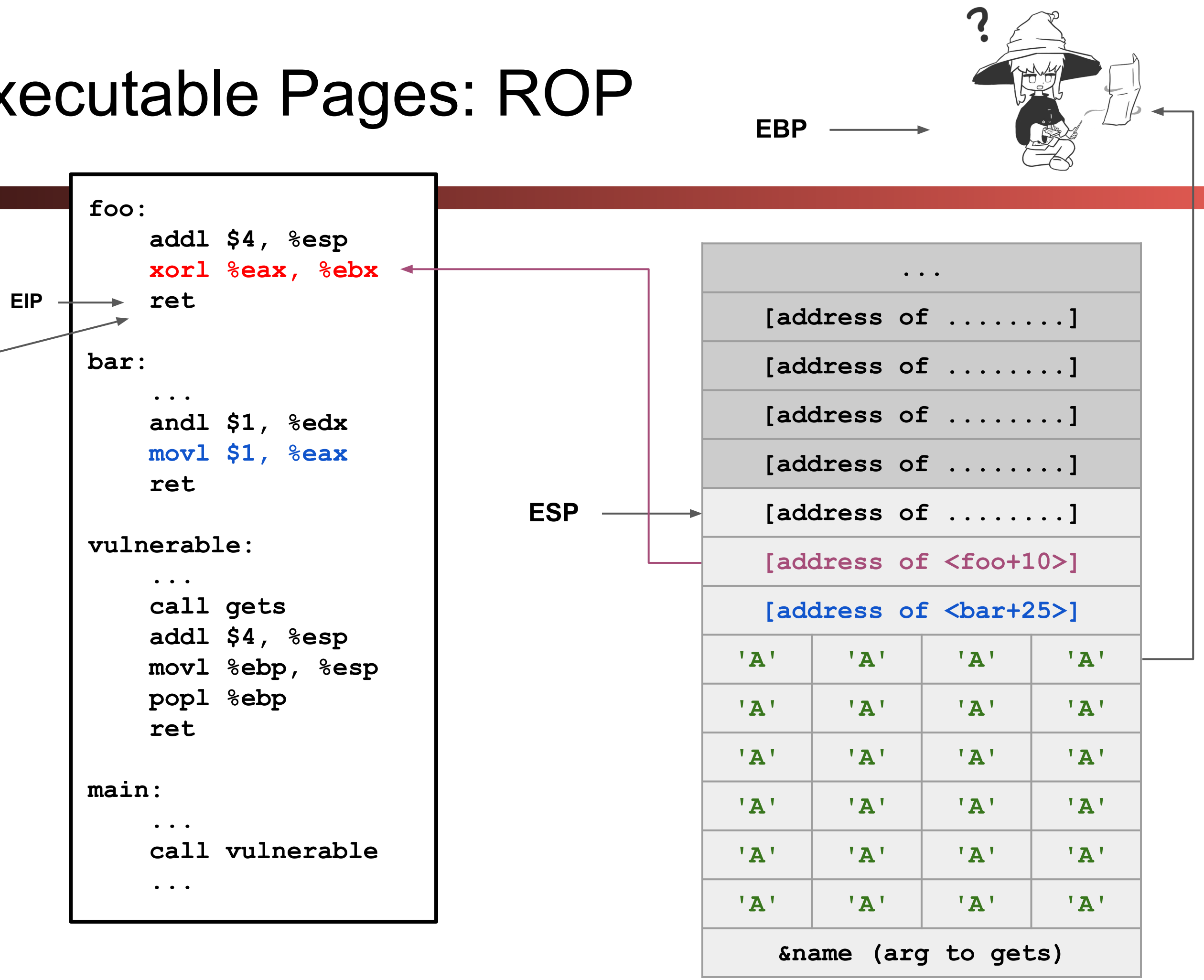
```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

...			
[address of ....]			
[address of ....]			
[address of ....]			
[address of ....]			
[address of ....]			
[address of <foo+10>]			
[address of <bar+25>]			
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
&name (arg to gets)			

Subverting Non-Executable Pages: ROP

The **`ret`** instruction always pops off the bottom of the stack, so execution continues based on the chain of addresses!

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```



```
foo:
    addl $4, %esp
    xorl %eax, %ebx
    ret

bar:
    ...
    andl $1, %edx
    movl $1, %eax
    ret

vulnerable:
    ...
    call gets
    addl $4, %esp
    movl %ebp, %esp
    popl %ebp
    ret

main:
    ...
    call vulnerable
    ...
```

Subverting Non-Executable Pages: ROP

- If the code base is big enough (imports enough libraries), there are usually enough gadgets in memory for you to run any shellcode you want
- **ROP compilers** can automatically generate a ROP chain for you based on a target binary and desired malicious code!
- Non-executable pages is not a huge issue for attackers nowadays
  - Having writable and executable pages makes an attacker's life easier, but not *that* much easier

Mitigation: Stack Canaries

Textbook Chapter 4.8 & 4.9

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
    - Mitigation: Stack canaries
4. Return from the function
5. Begin executing malicious shellcode
    - Mitigation: Non-executable pages

We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Analogy: Canary in a Coal Mine

- Miners protect themselves against toxic gas buildup in the mine with a canary
    - Canary: A small, noisy bird that is sensitive to toxic gas
    - If toxic gas builds up, the canary dies first
    - The miners notice that the canary has died and leave the mine
- The canary in the coal mine is a sacrificial animal
    - The miners don’t expect the canary to survive
    - However, the canary's death is a warning sign that saves the lives of the miners
- **Takeaway**: Let’s put a sacrificial value (a *canary*) on the stack
    - The value is not meaningful (we don’t care if it’s preserved)
    - The code never uses or changes this value
    - If the value changes, that's a warning sign that somebody is messing with our code!

Stack Canaries

- Idea: Add a sacrificial value on the stack, and check if it has been changed
  - When the program runs, generate a **random** secret value and save it in the canary storage
  - In the function prologue, place the canary value on the stack right below the SFP/RIP
  - In the function epilogue, check the value on the stack and compare it against the value in canary storage
- The canary value is never actually used by the function, so if it changes, somebody is probably attacking our system!

Stack Canaries: Properties

- A canary value is unique every time the program **runs** but the same for all functions **within a run**
- A canary value uses a NULL byte as the first byte to mitigate string-based attacks (since it terminates any string before it)
  - Example: A format string vulnerability with **`%s`** might try to print everything on the stack
    - The null byte in the canary will mitigate the damage by stopping the print earlier.

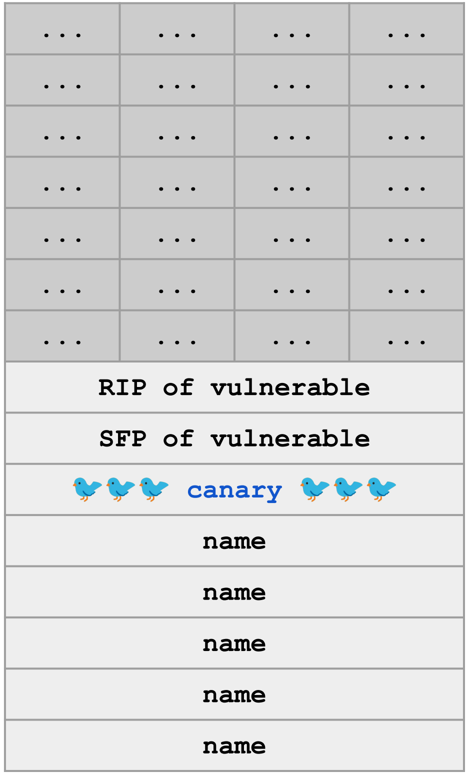
Stack Canaries

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

```
vulnerable:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    movl ($CANARY_ADDR), %eax # Load canary
    movl %eax, -4(%ebp)       # Save on stack

    ...

    movl -4(%ebp), %eax       # Load stack value
    cmpl %eax, ($CANARY_ADDR) # Compare to canary and...
    jne canary_failed         # ... crash if not equal
    movl %ebp, %esp
    popl %ebp
    ret
```

Because the write starts at name, the attacker has to overwrite the canary before the RIP or SFP!

Note: 20 bytes for **`name`** + 4 bytes for canary (32-bit architecture)



...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
RIP of vulnerable			
SFP of vulnerable			
🐦🐦🐦 canary 🐦🐦🐦			
name			
name			
name			
name			
name			

Stack Canaries: Efficiency

- Compiler inserts a few extra instructions, so there is more overhead
- In almost all applications, the performance impact is insignificant
    - Very cheap way to stop lots of common attacks!

Subverting Stack Canaries

- **Leak** the value of the canary: Overwrite the canary with itself
- **Bypass** the value of the canary: Use a random write, not a sequential write
- **Guess** the value of the canary: Brute-force

Subverting Stack Canaries: Leaking the Canary

- Any vulnerability that leaks stack memory could be used to leak the canary’s value
  - Example: Format string vulnerabilities let you print out values on the stack
- Once you learn the value of the stack canary, place it in the exploit such that the canary is overwritten with itself, so the value is unchanged!

Subverting Stack Canaries: Bypassing the Canary

- Stack canaries stop attacks that write to *increasing*, *consecutive* addresses *on the stack*
  - On the stack diagram: Writing upwards, with no gaps
  - Many common functions only write this way, e.g. **`gets`**, **`fgets`**, **`fread`**, etc.
- Stack canaries do not stop attacks that write to memory in other ways
  - An attacker can write *around* the canary
  - Example: Format string vulnerabilities let an attacker write to any location in memory
  - Example: Heap overflows never overwrite a stack canary (they write to the heap)
  - Example: C++ vtable exploits overwrite the vtable pointer without overwriting the canary

Subverting Stack Canaries: Guessing the Canary

- On 32-bit systems: 24 bits to guess
  - Remember that the first byte (8 bits) is always a NULL byte: 32 - 8 = 24
- On 64-bit systems: 56 bits to guess
  - 64 - 8 = 56
- Stack canaries are less effective on 32-bit systems since there are only 2^{24} possibilities (~16 million), which can feasibly be brute-forced

Subverting Stack Canaries: Guessing the Canary

- How feasible is guessing the canary?
    - It depends on your threat model
- How are you running the program?
    - If the program is running on your own computer, you can keep trying with nobody to stop you
    - If the program is running on a remote server, the server might see you sending the exploit over and over and reject your requests
- Does the program have a timeout?
    - Timeout: A mandatory waiting period after a failed request
    - No timeout: 10,000 tries per second = 2^{24} tries in around 30 minutes
    - 0.1 second timeout: 10 tries per second = 2^{24} tries in around 3 weeks
- More complicated timeouts are possible
    - 10 consecutive failures causes a 10-minute timeout: 1 try per minute = 2^{24} tries in ~32 years!
    - Exponentially growing timeout (the timeout doubles for each failure): 2^{24} tries is not happening

Mitigation: Pointer Authentication

Textbook Chapter 4.10

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
3. Overwrite the RIP with the address of the shellcode
    - Mitigation: Stack canaries
    - Mitigation: Pointer authentication
4. Return from the function
5. Begin executing malicious shellcode
    - Mitigation: Non-executable pages

We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Reminder: 32-Bit and 64-Bit Processors

- 32-bit processor: integers and pointers are 32 bits long
  - Can address 2^{32} bytes ≈ 4 GB of memory
- 64-bit processor: integers and pointers are 64 bits long
  - Can address 2^{64} bytes ≈ 18 exabytes ≈ 18 billion GB of memory
  - No modern computer can support this much memory
  - Even the best most modern computers only need 2^{42} bytes ≈ 4 terabytes ≈ 4000 GB of memory
  - At most 42 bits are needed to address all of memory
  - 22 bits are left unused (the top 22 bits in the address are always 0)

Pointer Authentication

- Recall stack canaries: A secret value stored in memory
  - If the secret value changes, detect an attack
  - One canary per function on the stack
- Idea: Instead of placing the secret value below the pointer, store a value in the pointer itself!
  - Use the unused bits in a 64-bit address to store a secret value
  - When storing a pointer in memory, replace the unused bits with a **pointer authentication code (PAC)**
  - Before using the pointer in memory, check if the PAC is still valid
    - If the PAC is invalid, crash the program
    - If the PAC is valid, restore the unused bits and use the address normally
  - Includes the RIP, SFP, any other pointers on the stack, and any other pointers outside of the stack (e.g. on the heap)

Pointer Authentication: Properties of the PAC

- Each possible address has its own PAC
    - Example: The PAC for the address `0x000000007ffffec0` is different from the PAC for `0x000000007ffffec4`
    - If an attacker changes the address without changing the PAC, the PAC will no longer be valid
- Only someone who knows the CPU's master secret can generate a PAC for an address
    - An attacker cannot generate a PAC for their malicious pointer without the master secret
    - An attacker cannot generate a PAC using a PAC for a different address
    - Later: We'll discuss how this algorithm works (MACs in the cryptography unit)
- The CPU's master secret is not accessible to the program
    - Leaking program memory will not leak the master secret
        - Contrast with canaries, which can be leaked

Subverting Pointer Authentication

- Find a vulnerability to trick the program to generating a PAC for any address
- Learn the master secret
  - The operating system has to set up the secrets: What if there is a vulnerability in the OS?
  - Workaround: Embed the master secret in the CPU, which can only be used to generate PACs, never read directly
- Guess a PAC: Brute-force
  - Most 64-bit systems use 32-52 bits (plus one extra configuration bit) for addressing, which leaves 11-31 bits for the PAC
  - Number of possible PACs is in the millions or billions, so possibly feasible depending on your threat model
- Pointer reuse
  - If the CPU already generated another PAC for another pointer, we can copy that pointer and use it elsewhere

Defenses Against Pointer Reuse

- In practice, there are usually multiple master secrets for different types of pointers
    - ARM uses 5 master secrets: 2 instruction pointer secrets (IA and IB), 2 data pointer secrets (DA and DB), and 1 general-purpose secret (GA)
    - Instruction pointer secrets are used for pointers to machine instructions (e.g. RIP)
    - Data pointer secrets are used for pointers to data (e.g. local variables)
    - Data pointers can’t be reused as instruction pointers, and vice-versa
- The CPU can generate a unique PAC for each pointer and “context”
    - Context: usually the address where the pointer is located
    - The same pointer will have a different PAC depending on where in memory it's located
    - If an attacker copies a pointer and PAC to a different location, the PAC is no longer valid!

Pointer Authentication on ARM

- Pointer authentication is supported by:
    - ARM 8.3 (a new architecture, like x86 or RISC-V)
    - The latest Apple chips (starting with the A12 and including the new M1), which use ARM
    - macOS on ARM (operating system)
- Probably the biggest benefit for Apple going to ARM
    - Can take advantage of the more efficient instructions instead of backwards-compatible ones
    - Usable in both standard user programs and kernel programs (privileged code run by the OS)
- x86 has not developed a similar defense

Mitigation: Address Space Layout Randomization (ASLR)

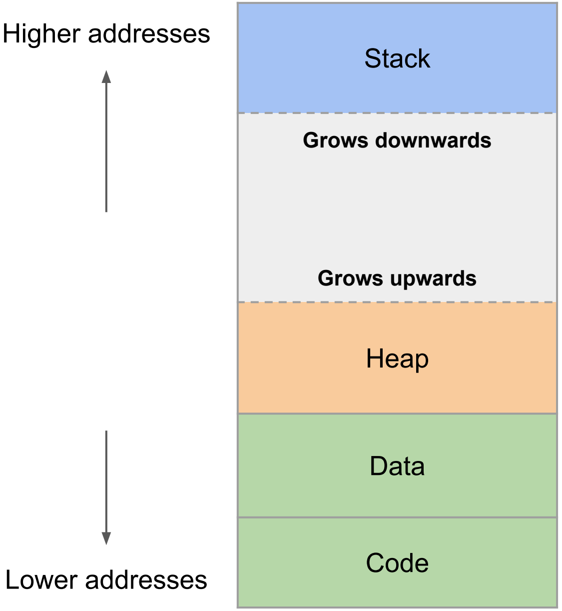
Textbook Chapter 4.11 & 4.12

Recall: Putting Together an Attack

1. Find a memory safety (e.g. buffer overflow) vulnerability
2. Write malicious shellcode at a known memory address
    - Mitigation: Address-space layout randomization
3. Overwrite the RIP with the address of the shellcode
    - Mitigation: Stack canaries
    - Mitigation: Pointer authentication
4. Return from the function
5. Begin executing malicious shellcode
    - Mitigation: Non-executable pages

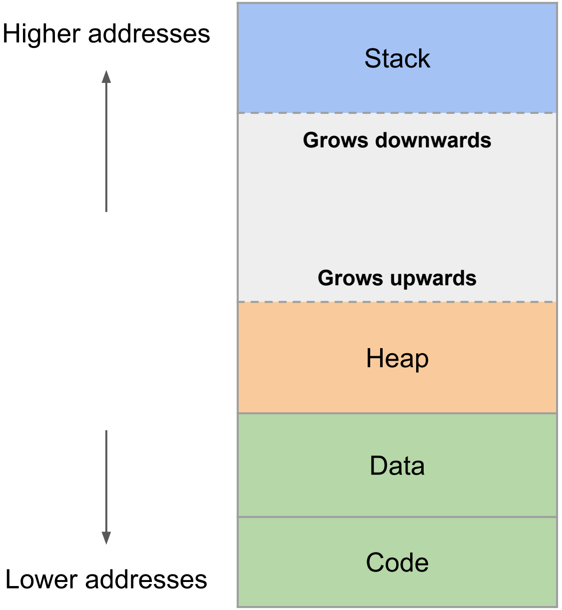
We can defend against memory safety vulnerabilities by making each of these steps more difficult (or impossible)!

Recall: x86 Memory Layout

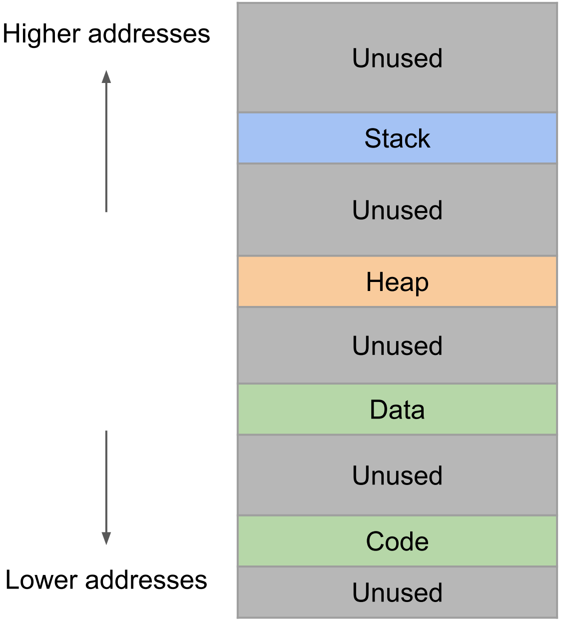


In theory, x86 memory layout looks like this...

Recall: x86 Memory Layout

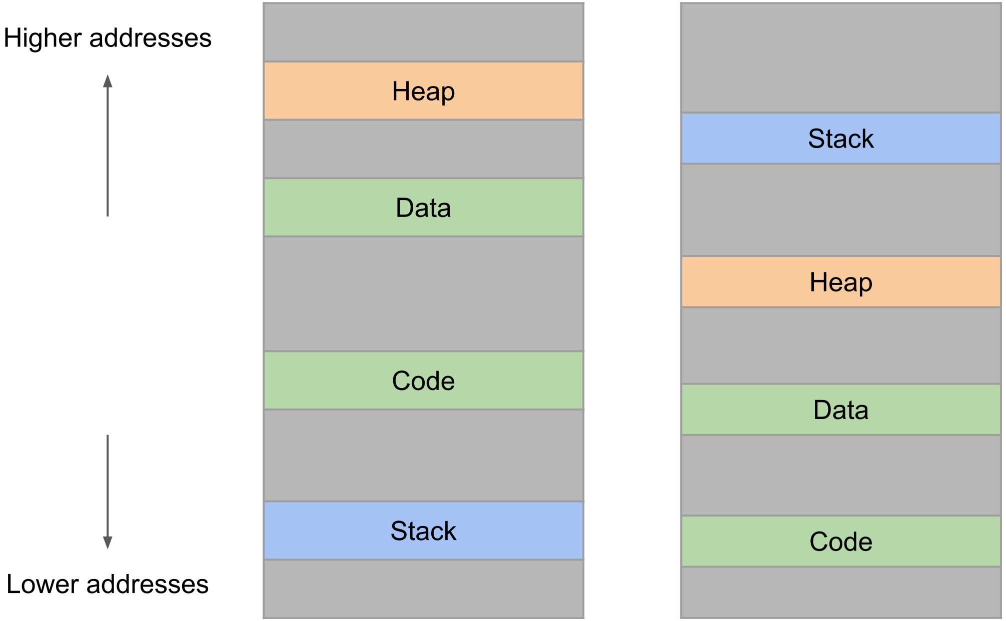


In theory, x86 memory layout looks like this...



...but in practice, it usually looks like this (mostly empty)!

Recall: x86 Memory Layout



Idea: Put each segment of memory in a different location each time the program is run

Address Space Layout Randomization

- **Address space layout randomization (ASLR)**: Put each segment of memory in a different location each time the program is run
    - The attacker can’t know where their shellcode will be because its address changes every time you run the program
- ASLR can shuffle all four segments of memory
    - Randomize the stack: Can’t place shellcode on the stack without knowing the address of the stack
    - Randomize the heap: Can’t place shellcode on the heap without knowing the address of the heap
    - Randomize the code: Can’t construct a ROP chain or return-to-libc attack without knowing the address of code
    - Within each segment of memory, relative addresses are the same (e.g. the RIP is always 4 bytes above the SFP)

ASLR: Efficiency

- Recall from 61C
    - Programs are dynamically linked at runtime
    - We already have to do the work of going through the executable and rewriting code to contain known addresses before executing it
- ASLR has effectively no overhead, since we have to do relocation anyway!

Subverting ASLR

- Leak the address of a pointer, whose address relative to your shellcode is known
  - Relative addresses are usually fixed, so this is sufficient to undo randomization!
  - Leak a stack pointer: Leak the location of the stack
  - Leak an RIP: Leak the location of the caller
- Guess the address of your shellcode: Brute-force
  - Randomization usually happens on page boundaries (usually 12 bits for 4 KiB pages)
  - 32-bit: 32 - 12 = 20 bits, 2^{20} possible pages, which is feasibly brute-forced
  - 64-bit (usually 48-bit addressing): 48 - 12 = 36 bits, 2^{36} possible pages

Relative Addresses

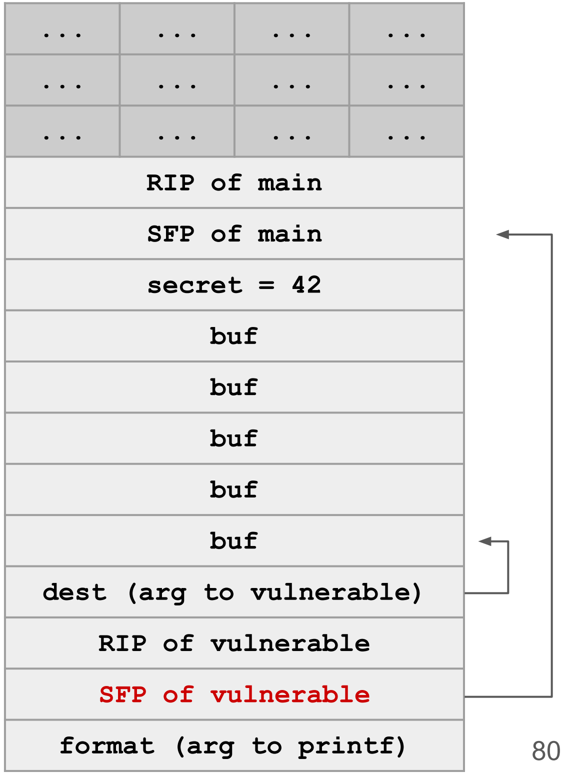
We know that the SFP is a pointer to the stack. How would you print the value of the SFP?

Input:
`'%x'`

If the output is `bfff0408` what is the address of **`secret`**?

`secret` is 4 bytes below where the SFP points, so its address is `0xbfff0404`!

```
void vulnerable(char *dest) {
    // Format string vulnerability
    printf(dest);
}

int main(void) {
    int secret = 42;
    char buf[20];
    fgets(buf, 20, stdin);
    vulnerable(buf);
}
```



Combining Mitigations

Textbook Chapter 4.13

Combining Mitigations

- Recall: We can use multiple mitigations together
    - Synergistic protection: one mitigation helps strengthen another mitigation
    - Force the attacker to find multiple vulnerabilities to exploit the program
    - Defense in depth
- Example: Combining ASLR and non-executable pages
    - An attacker can't write their own shellcode, because of non-executable pages
    - An attacker can't use existing code in memory, because they don't know the addresses of those code (ASLR)
- To defeat ASLR *and* non-executable pages, the attacker needs to find two vulnerabilities
    - First, find a way to leak memory and reveal the address randomization (defeat ASLR)
    - Second, find a way to write to memory and write a ROP chain (defeat non-executable pages)

Combining Mitigations

- Memory safety defenses used by Apple iOS
  - ASLR is used for user programs (apps) and kernel programs (operating system programs)
  - Non-executable pages are used whenever possible
  - Applications are sandboxed to limit the damage of an exploit (TCB is the operating system)
- Trident exploit
  - Developed by the NSO group, a spyware vendor, to exploit iPhones
  - Exploit Safari with a memory corruption vulnerability → execute arbitrary code in the sandbox
  - Exploit another vulnerability to read the kernel stack (operating system memory in the sandbox)
  - Exploit another vulnerability in the kernel (operating system) to execute arbitrary code
- **Takeaway**: Combining mitigations forces the attacker to find multiple vulnerabilities to take over your program. The attacker's job is harder, but not impossible!

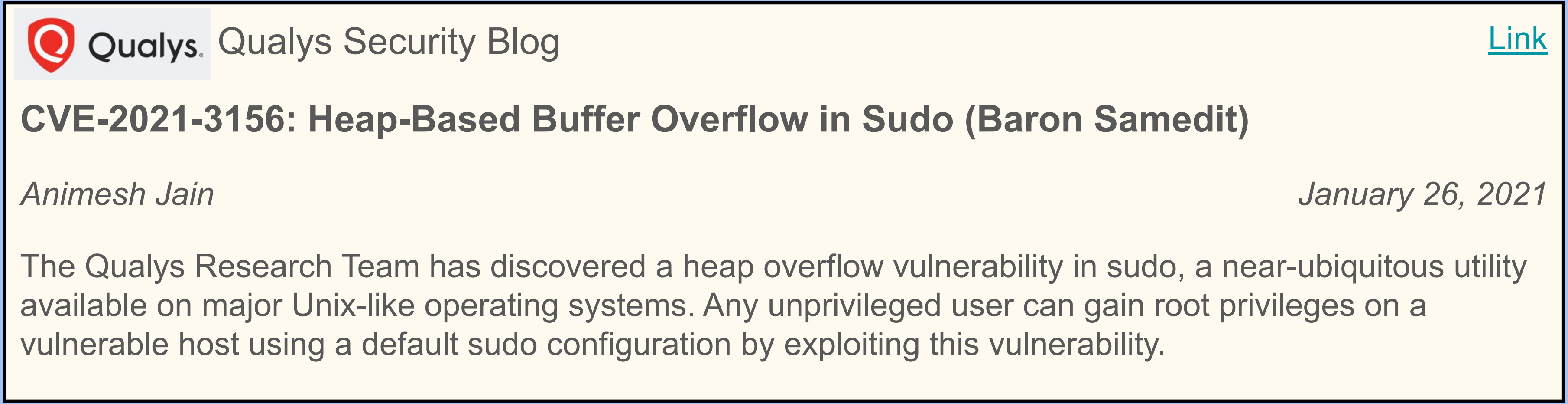
Enabling Mitigations

- Many mitigations (stack canaries, non-executable pages, ASLR) are effectively free today (insignificant performance impact)
- The programmer sometimes has to manually enable mitigations
  - Example: Enable ASLR and non-executable pages when running a program
  - Example: Setting a flag to compile a program with stack canaries
- If the default is disabling the mitigation, the default will be chosen
  - Recall: Consider human factors!
  - Recall: Use fail-safe defaults!

Enabling Mitigations: CISCO

- Cisco’s Adaptive Security Appliance (ASA)
    - Cisco: A major vendor of technology products (one of 30 giant companies in the Dow Jones stock index)
    - ASA: A network security device that can be installed to protect an entire network (e.g. AirBears2)
- Mitigations used by the ASA
    - No stack canaries
    - No non-executable pages
    - No ASLR
    - Easy for the NSA (or other attackers) to exploit!
- **Takeaway**: Even major companies can forget to enable mitigations. Always enable memory safety mitigations!

Enabling Mitigations: Internet of Things



Qualys Security Blog

Link

CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit)

Animesh Jain

January 26, 2021

The Qualys Research Team has discovered a heap overflow vulnerability in sudo, a near-ubiquitous utility available on major Unix-like operating systems. Any unprivileged user can gain root privileges on a vulnerable host using a default sudo configuration by exploiting this vulnerability.

Takeaway: Many (most?) IoT devices don’t enable basic mitigations

Summary: Memory Safety Mitigations

- Memory-safe languages
  - Using a memory-safe language (e.g. Python, Java) stops all memory safety vulnerabilities.
  - Why use a non-memory-safe language?
    - Commonly-cited reason, but mostly a myth: Performance
    - Real reason: Legacy, existing code
- Writing memory-safe code
  - Carefully write and reason about your code to ensure memory safety in a non-memory-safe language
  - Requires programmer discipline, and can be tedious sometimes
- Building secure software
  - Use tools for analyzing and patching insecure code
  - Test your code for memory safety vulnerabilities
  - Keep any external libraries updated for security patches

Summary: Memory Safety Mitigations

- Mitigation: **Non-executable pages**
    - Make portions of memory either executable or writable, but not both
    - Defeats attacker writing shellcode to memory and executing it
    - Subversions
        - **Return-to-libc**: Execute an existing function in the C library
        - **Return-oriented programming** (**ROP**): Create your own code by chaining together small gadgets in existing library code
- Mitigation: **Stack canaries**
    - Add a sacrificial value on the stack. If the canary has been changed, someone's probably attacking our system
    - Defeats attacker overwriting the RIP with address of shellcode
    - Subversions
        - An attacker can write around the canary
        - The canary can be leaked by another vulnerability (e.g. format string vulnerability)
        - The canary can be brute-forced by the attacker

Summary: Memory Safety Mitigations

- Mitigation: **Pointer authentication**
    - When storing a pointer in memory, replace the unused bits with a pointer authentication code (PAC). Before using the pointer in memory, check if the PAC is still valid
    - Defeats attacker overwriting the RIP (or any pointer) with address of shellcode
- Mitigation: **Address space layout randomization** (**ASLR**)
    - Put each segment of memory in a different location each time the program is run
    - Defeats attacker knowing the address of shellcode
    - Subversions
        - Leak addresses with another vulnerability
        - Brute-force attack to guess the addresses
- Combining mitigations
    - Using multiple mitigations usually forces the attacker to find multiple vulnerabilities to exploit the program (defense-in-depth)