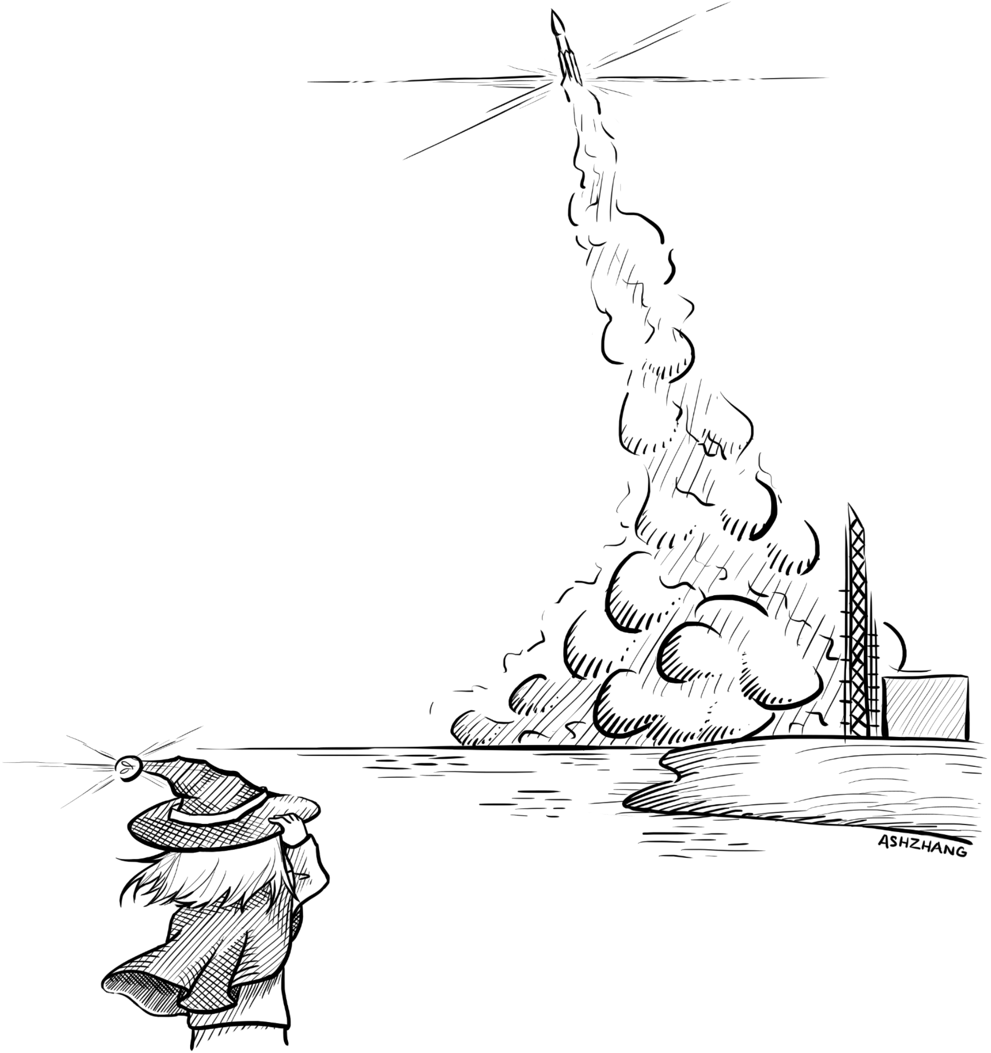
# Mitigating Memory Safety Vulnerabilities

## CS 161 Spring 2022 - Lecture 5

# Announcements

- Project 1 is released
    - Checkpoint is due Friday, February 4th, 11:59 PM PT
    - Final submission is due Friday, February 18th, 11:59 PM PT



# Writing Robust Exploits

# Variance in Exploits

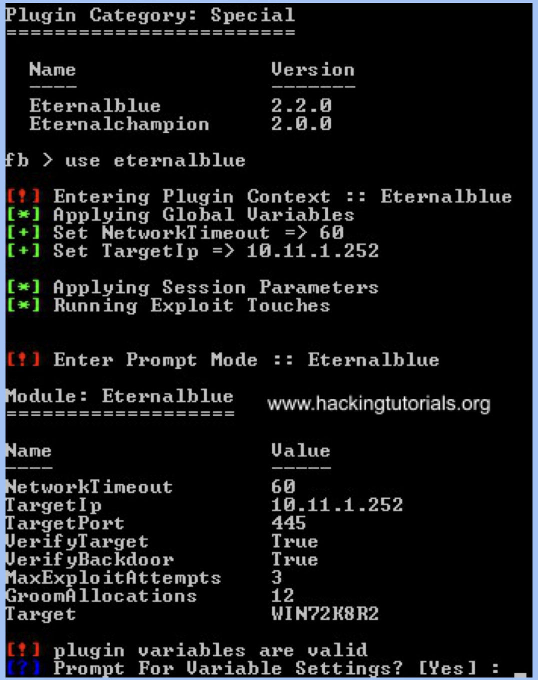
- Exploits can often be very brittle
  - A working exploit can depend on operating system, memory layout, time, etc.
  - You will see this in Project 1: Your exploit files will not work on someone else’s virtual machine (VM) because your memory layout is different
- Each machine or program might require its own set of constants (“magic numbers”) to product a working exploit
- Making an exploit robust is an art unto itself

# EXTRABACON

- An NSA exploit for Cisco Adaptive Security Appliances (ASA)
  - It had an exploitable stack-overflow vulnerability in its network monitoring protocol (SNMP)
- Making the exploit robust required two steps:
  - Query for the version of ASA that was being used using SNMP
  - Select the correct set of magic numbers based on the version returned

# ETERNALBLUE(screen)

- Another NSA exploit stolen by the same group that stole EXTRABACON
  - Exploited a vulnerability in Windows file sharing (SMBv1)
- It was a very robust exploit
  - Lead to malware like Wana Decrypt0r (Wanacry)
  - Used by the NSA to steal foreign intelligence
- But before it reached that point, it would crash computers more often than exploiting them
  - It was nicknamed “ETERNALBLUESCREEN”
- **Takeaway**: Writing a robust exploit is very difficult (we won’t ask you to do this)

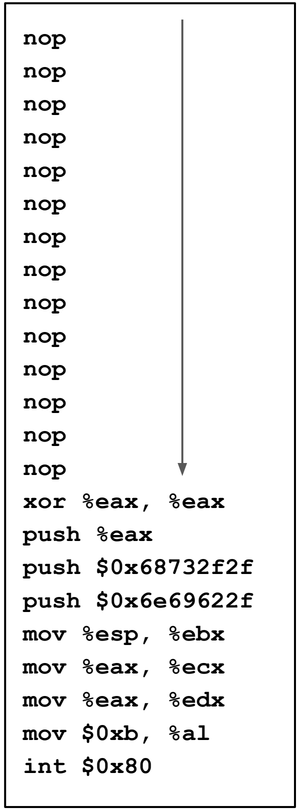


```
Plugin Category: Special
========================

  Name                 Version
  ----                 -------
  Eternalblue          2.2.0
  Eternalchampion      2.0.0

fb > use eternalblue

[!] Entering Plugin Context :: Eternalblue
[*] Applying Global Variables
[+] Set NetworkTimeout => 60
[+] Set TargetIp => 10.11.1.252

[*] Applying Session Parameters
[*] Running Exploit Touches


[!] Enter Prompt Mode :: Eternalblue

Module: Eternalblue
===================

Name                 Value
----                 -----
NetworkTimeout       60
TargetIp             10.11.1.252
TargetPort           445
VerifyTarget         True
VerifyBackdoor       True
MaxExploitAttempts   3
GroomAllocations     12
Target               WIN72K8R2

[!] plugin variables are valid
[?] Prompt For Variable Settings? [Yes] : _
```

www.hackingtutorials.org

# NOP Sleds

- Idea: Instead of having to jump to an exact address, make it “close enough” so that small shifts don’t break your exploit
- NOP: Short for no-operation or no-op, an instruction that does nothing (except advance the EIP)
  - A real instruction in x86, unlike RISC-V
  - RISC-V instead uses **`addi x0 x0 0`** as a no-op
- Chaining a long sequence of NOPs means that landing anywhere in the sled will bring you to your shellcode



```
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
xor %eax, %eax
push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp, %ebx
mov %eax, %ecx
mov %eax, %edx
mov $0xb, %al
int $0x80
```

# Next: Memory Safety Mitigations

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

# 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.

# 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

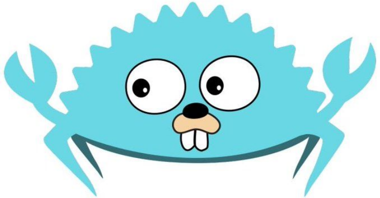
- **Memory-safe languages** are designed to check bounds and prevent undefined memory accesses
- By design, memory-safe languages are not vulnerable to memory safety vulnerabilities
  - Using a memory-safe language is the *only* way to stop 100% of memory safety vulnerabilities
- Examples: Java, Python, 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
- Main problem with garbage collectors is not their performance, but the fact that they run at *unpredictable* times
  - Total runtime of a C program is effectively unchanged if you replace **`malloc`**/**`free`** with a “conservative garbage collector”

# 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!
  - If a garbage collector's pauses don't matter, use Go (*very* good support for parallel programming on multicore systems)
  - If a garbage collector's pauses do matter, or you do need to do unsafe things, use Rust



Mashup of Go and Rust mascots.

# 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
  - *Amdahl's law for programmer time*: The lifetime execution time of your program is really the execution time for all executions + the time it took for you to write it in the first place!
- “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
- 2014: Swift, a new memory-safe language, introduced
    - But still a heck of a lot in 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
  - Implement a reference monitor
  - 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

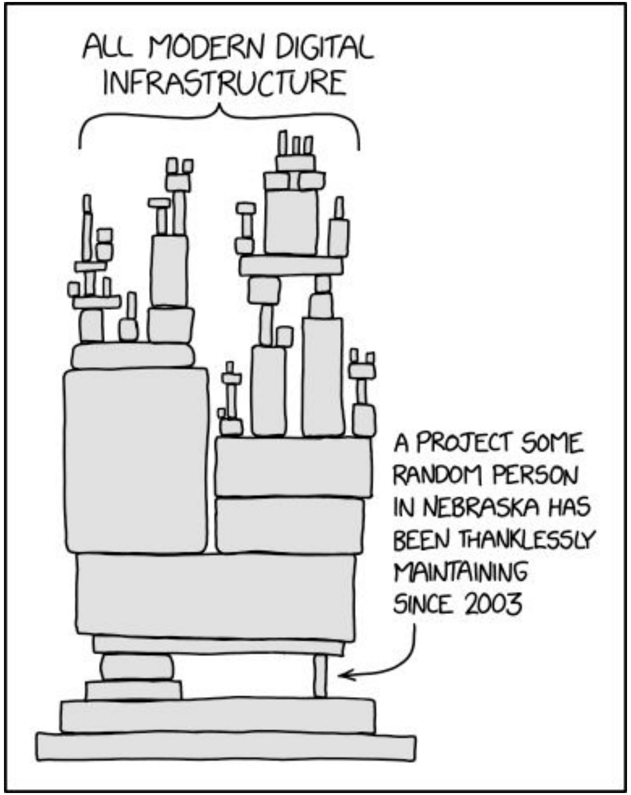
- Bug-finding tools
  - Excellent resource, as long as there aren’t too many false bugs
  - Too many false bugs = wasted programmer time
  - Example: Valgrind is particularly useful for C memory bugs
- Code review
  - Hiring someone to look over your code for memory safety errors
  - Can be very effective… but also expensive
- Vulnerability scanning
  - Probe your systems for known flaws
- Penetration testing (“pen-testing”)
  - Pay someone to break into your system
  - Take notes on how they did it

# Testing for Software Security Issues

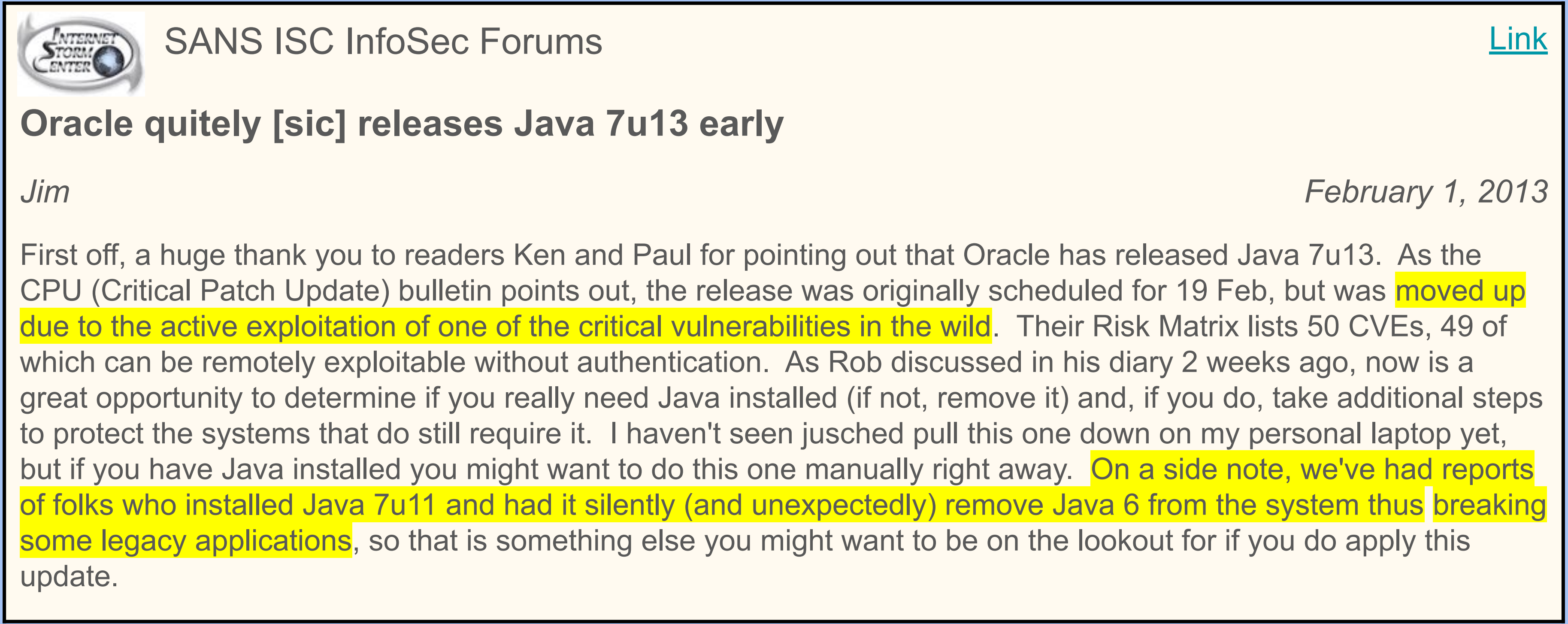
- What makes testing a program for security problems difficult?
  - We're testing for the *absence* of vulnerabilities
  - Normal inputs rarely reveal security vulnerabilities
- How can we test programs for memory safety vulnerabilities?
  - Fuzz testing: Random inputs
  - Use tools like Valgrind (tool 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

# The Software Supply Chain

- 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
  - Management burden (the “patch treadmill” never stops)
- A big and growing problem: Malicious updates
  - An attacker takes over or buys an open source library


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

# Patching Systems



SANS ISC InfoSec Forums

Link

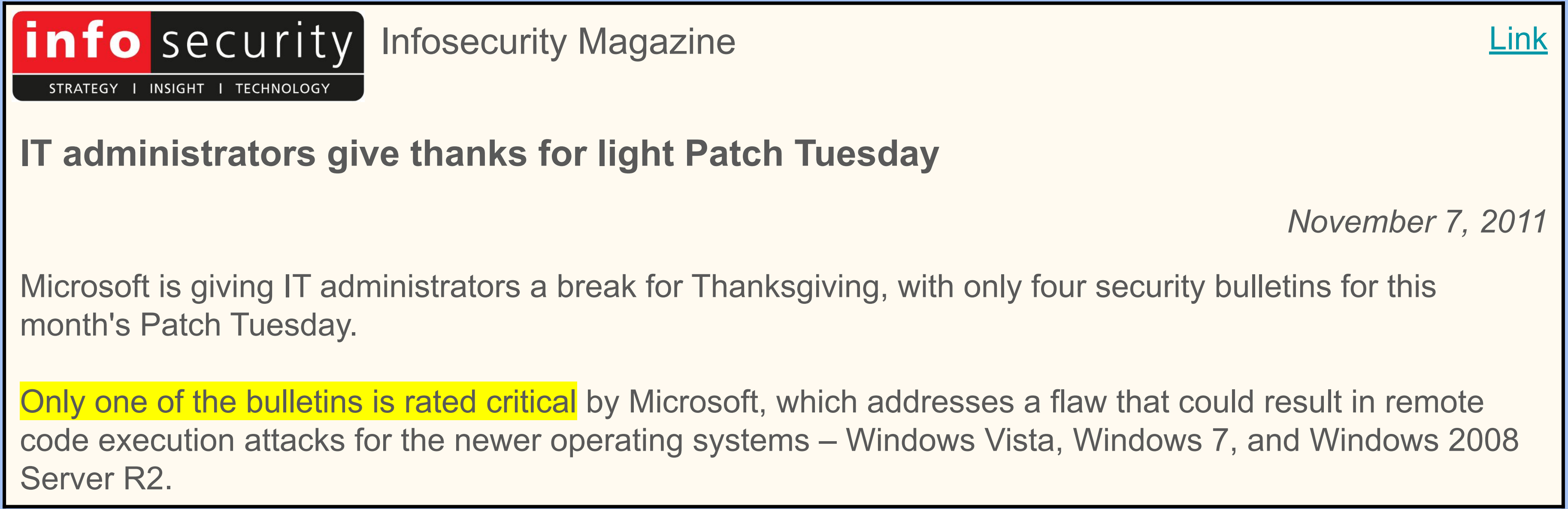
**Oracle quitely [sic] releases Java 7u13 early**

*Jim* *February 1, 2013*

First off, a huge thank you to readers Ken and Paul for pointing out that Oracle has released Java 7u13. As the CPU (Critical Patch Update) bulletin points out, the release was originally scheduled for 19 Feb, but was moved up due to the active exploitation of one of the critical vulnerabilities in the wild. Their Risk Matrix lists 50 CVEs, 49 of which can be remotely exploitable without authentication. As Rob discussed in his diary 2 weeks ago, now is a great opportunity to determine if you really need Java installed (if not, remove it) and, if you do, take additional steps to protect the systems that do still require it. I haven't seen jusched pull this one down on my personal laptop yet, but if you have Java installed you might want to do this one manually right away. On a side note, we've had reports of folks who installed Java 7u11 and had it silently (and unexpectedly) remove Java 6 from the system thus breaking some legacy applications, so that is something else you might want to be on the lookout for if you do apply this update.

**Takeaway**: Security patches can break older systems

# Patching Systems



Infosecurity Magazine

Link

### IT administrators give thanks for light Patch Tuesday

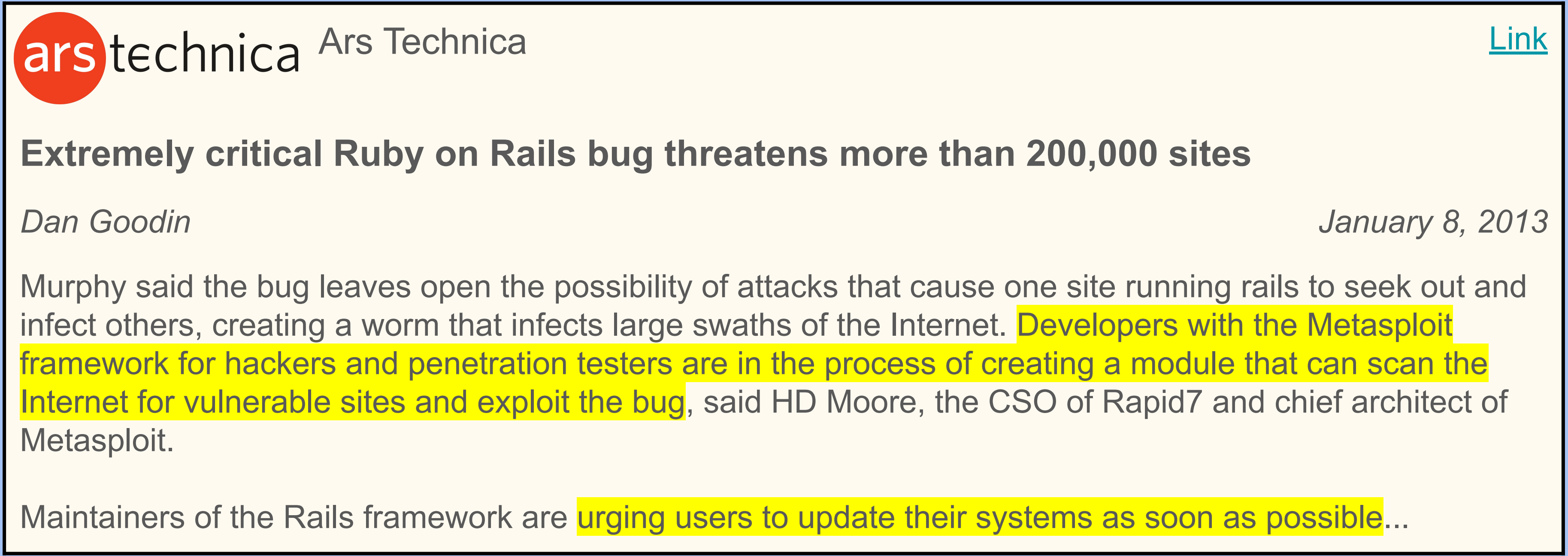
*November 7, 2011*

Microsoft is giving IT administrators a break for Thanksgiving, with only four security bulletins for this month's Patch Tuesday.

Only one of the bulletins is rated critical by Microsoft, which addresses a flaw that could result in remote code execution attacks for the newer operating systems – Windows Vista, Windows 7, and Windows 2008 Server R2.

**Takeaway**: Security patches are released frequently

# Patching Systems



ars technica Ars Technica

Link

**Extremely critical Ruby on Rails bug threatens more than 200,000 sites**

*Dan Goodin* *January 8, 2013*

Murphy said the bug leaves open the possibility of attacks that cause one site running rails to seek out and infect others, creating a worm that infects large swaths of the Internet. Developers with the Metasploit framework for hackers and penetration testers are in the process of creating a module that can scan the Internet for vulnerable sites and exploit the bug, said HD Moore, the CSO of Rapid7 and chief architect of Metasploit.

Maintainers of the Rails framework are urging users to update their systems as soon as possible...

**Takeaway**: Vulnerabilities become exploits quickly. It’s crucial to patch as soon as possible!

# Building a “New to Project Checklist”

- Turn on *all* compiler warnings
    - Example: **`-Wformat=2`** on gcc
    - Compilers can warn on many memory safety errors
- Maintain a list of unsafe functions to search for
    - Examples: **`gets()`**, **`sprintf()`**
    - When you find them, fix/replace with the safer alternatives
- Make sure to check for memory errors in your testing infrastructure
    - Example: Run all tests in Valgrind, which detects many C memory problems when they occur
- Look for serialization vulnerabilities
    - Examples: **`serialize()`** in Java or **`pickle`** in Python
    - Refactor the code to use safer alternatives like JSON

# 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)

# Exploit Mitigations

- Mitigations involve a large back-and-forth arms race
  - Security researchers find a new mitigation to make an exploit harder
  - Attackers find a way to defeat the mitigation
- Mitigations make attacks harder, but not impossible
  - The only way to prevent all buffer overflow attacks is to use a memory-safe language
- Mitigations can act synergistically
  - Multiple defenses work better together than apart: Adds defense in depth

# 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: 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

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 (`\x00`) 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.
  - Example: `strcpy` will stop when it hits a NULL byte

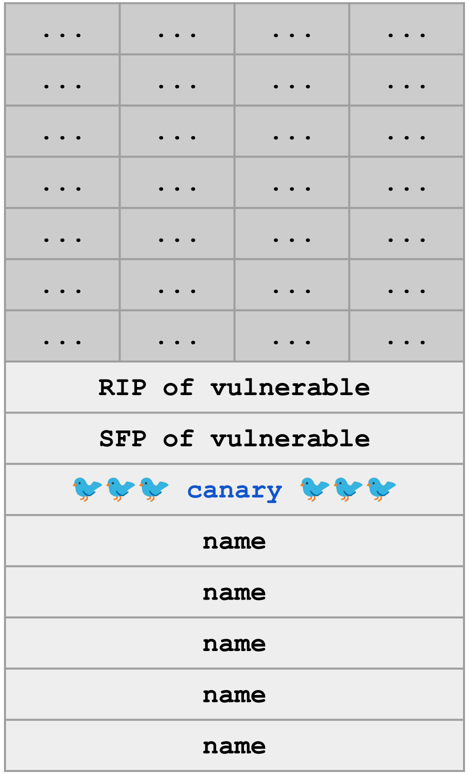
# Stack Canaries

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

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)

```
vulnerable:
    push %ebp
    mov %esp, %ebp
    sub $24, %esp
    mov ($CANARY_ADDR), %eax # Load canary
    mov %eax, -4(%ebp)       # Save on stack

    ...

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



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

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

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

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

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

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

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

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 location 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!

*cough* project question 3 *cough*

# 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: 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
    - 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)!

# 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
  - Calling a function is just jumping to it, so returning to the *beginning* of another function means it will look for arguments 4 bytes above the current ESP
- Consider the **`system`** function, which executes a shell command. We want to execute it like this:

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

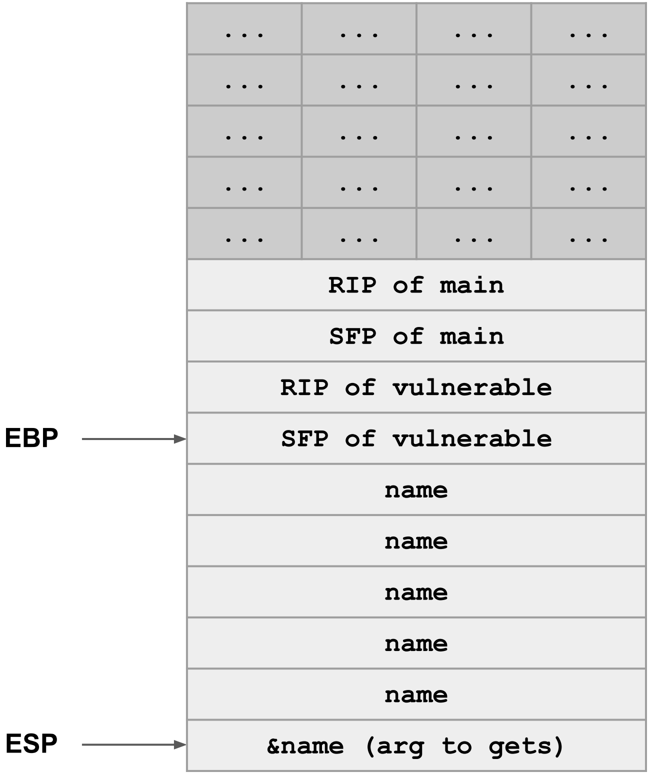
# Subverting Non-Executable Pages: Return-to-libc

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
          add $4, %esp
          mov %ebp, %esp
          pop %ebp
          ret

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



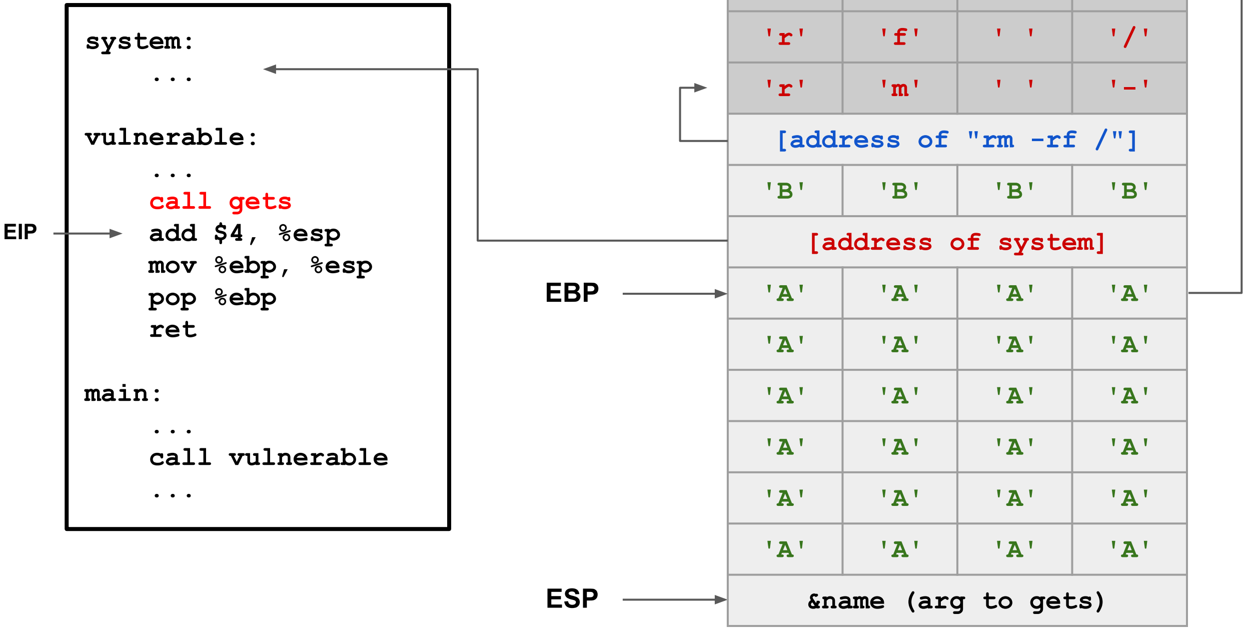
```
            ... ... ... ...
            ... ... ... ...
            ... ... ... ...
            ... ... ... ...
            ... ... ... ...
            RIP of main
            SFP of main
            RIP of vulnerable
EBP  →      SFP of vulnerable
            name
            name
            name
            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;
}
```

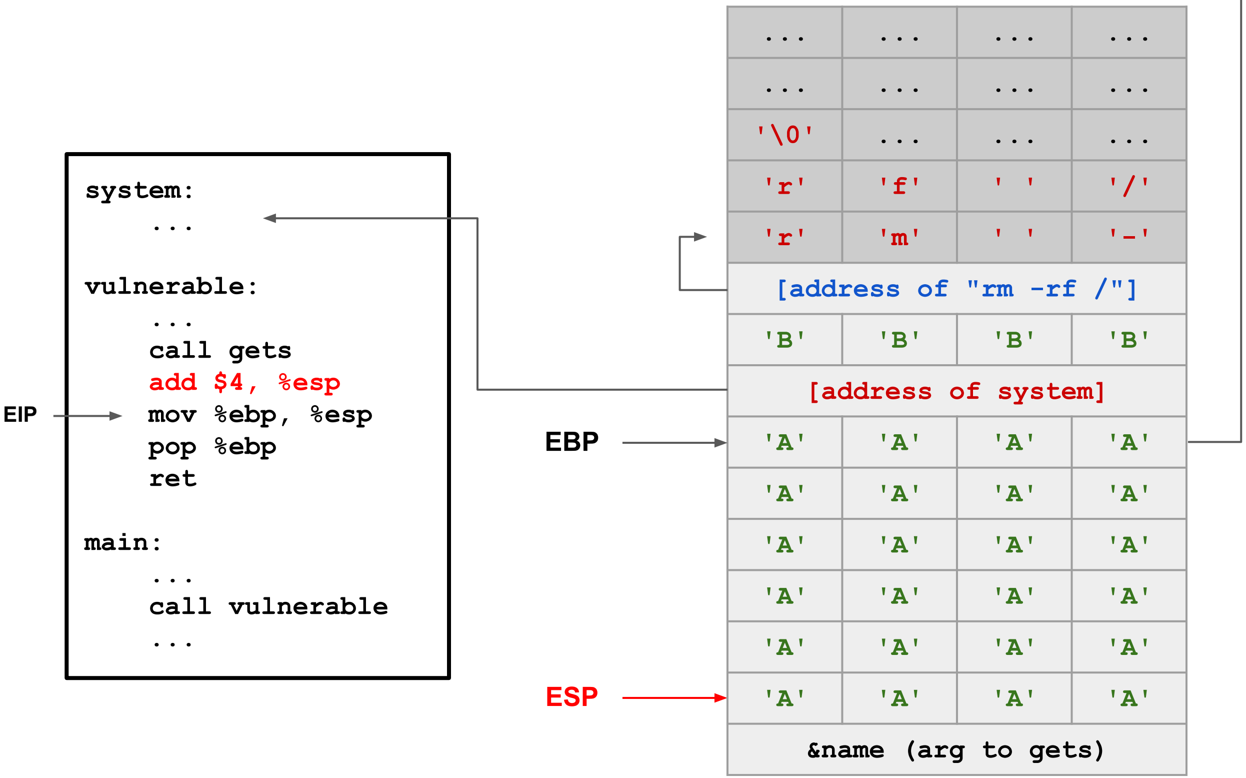


```
system:
    ...

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %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;
}
```



EIP

```
system:
    ...

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

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

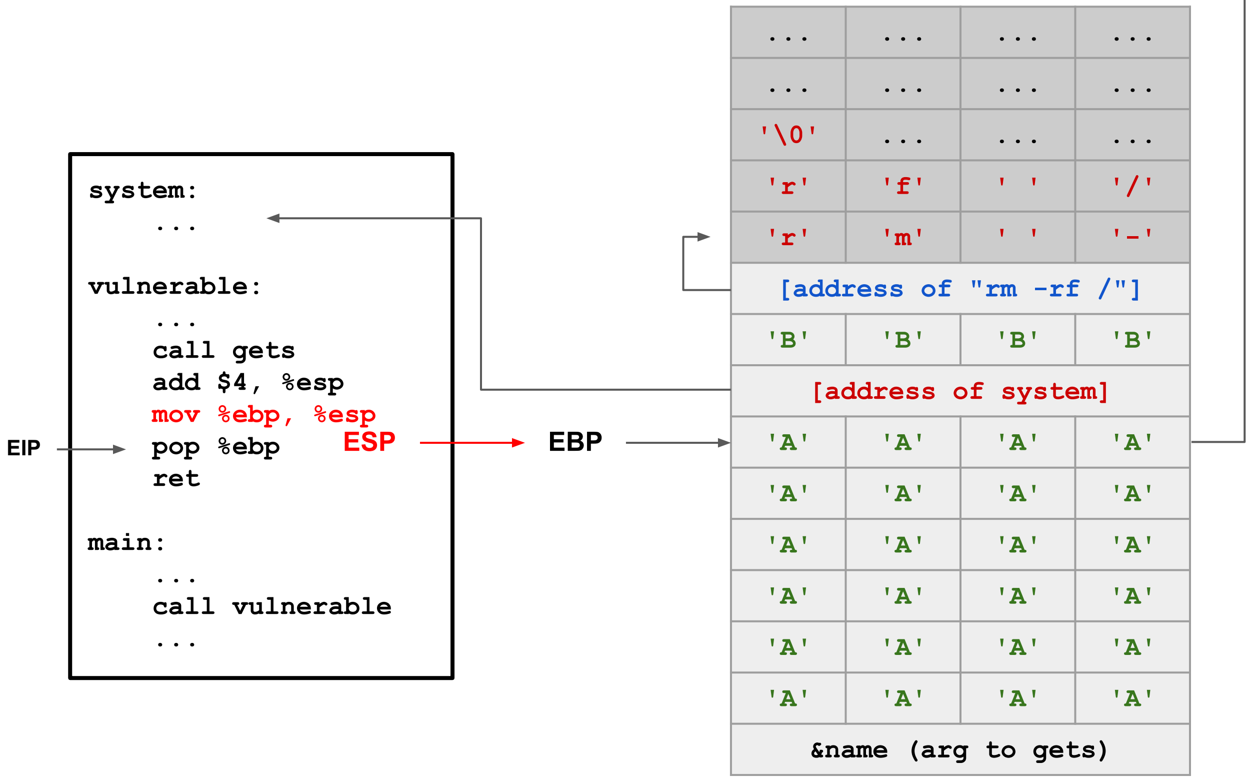
EBP

ESP

| | | | |
|---|---|---|---|
| ... | ... | ... | ... |
| ... | ... | ... | ... |
| '\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;
}
```



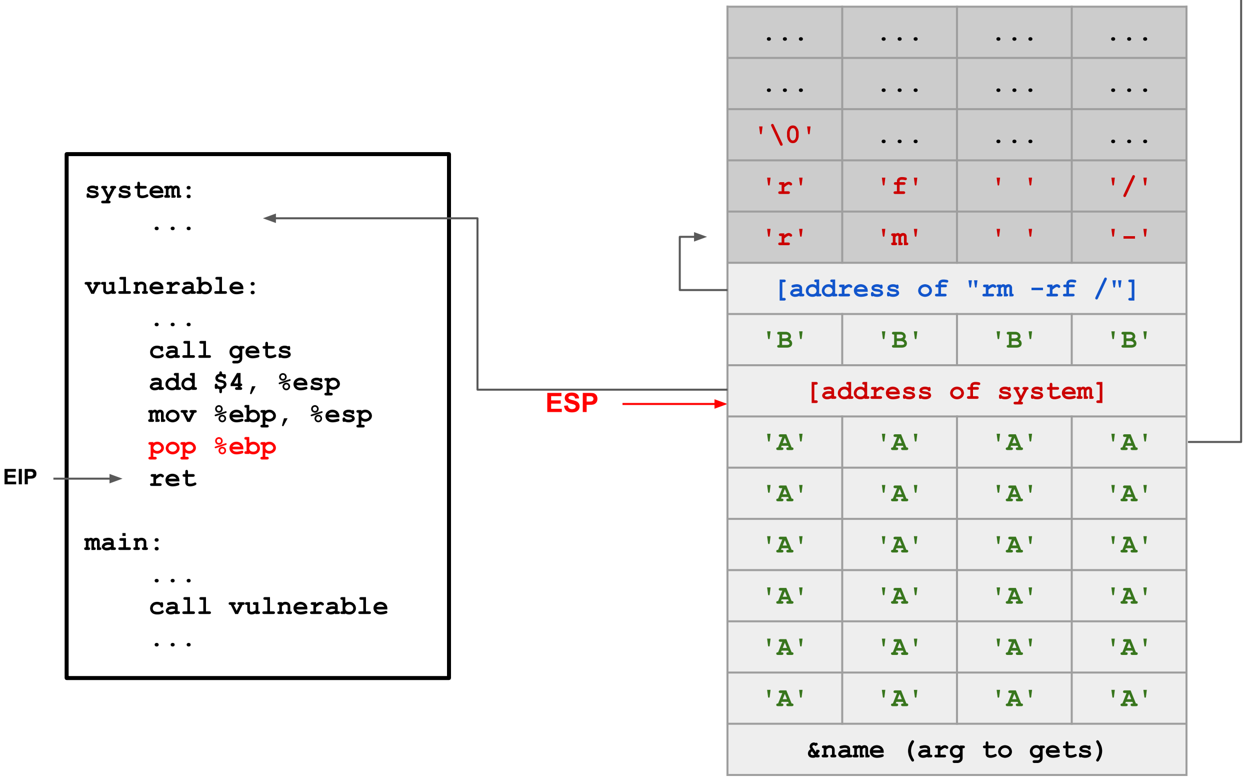
```
system:
    ...

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %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

EBP

```
int system(char *command);

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

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



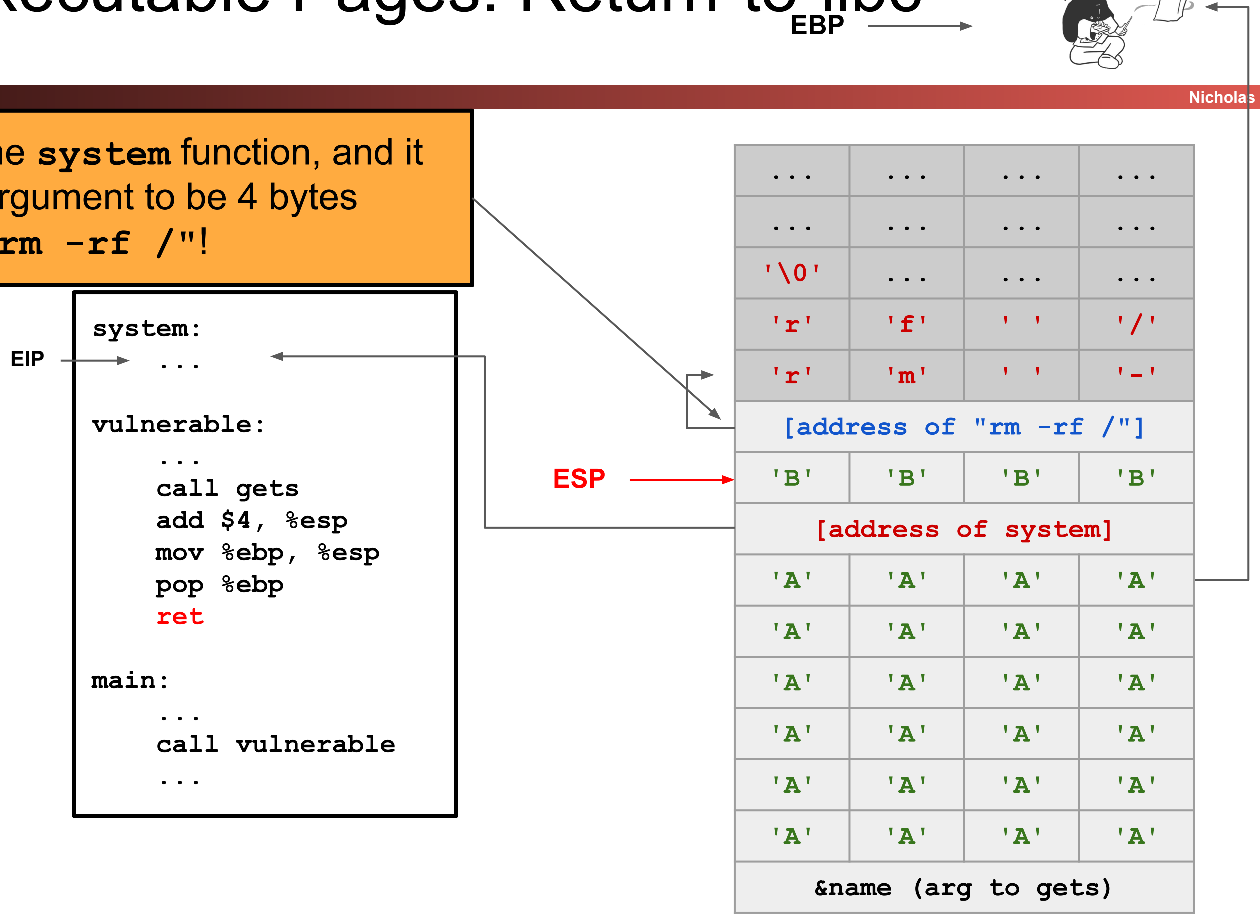
```
system:
    ...

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %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;
}
```



```
system:
    ...

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %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

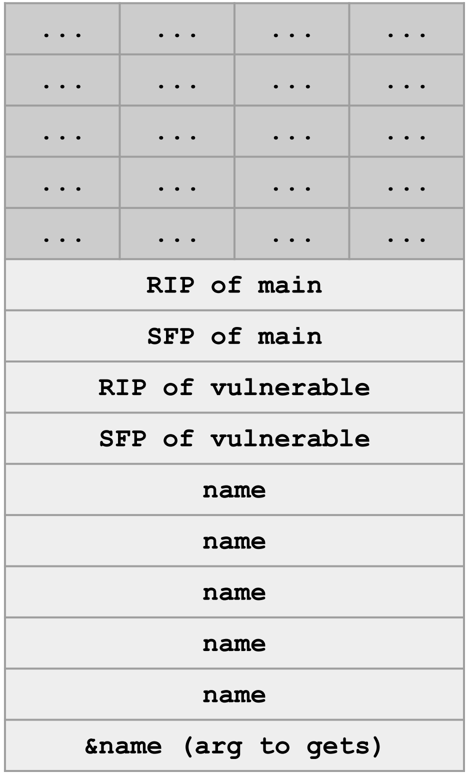
Example: Let’s say our shellcode involves the following sequence:

```
mov $1, %eax
xor %eax, %ebx
```

The following is present in memory:

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

bar:
          ...
<bar+22> and $1, %edx
<bar+25> mov $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:

```
mov $1, %eax
xor %eax, %ebx
```

The following is present in memory:

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

bar:
      ...
<bar+22> and $1, %edx
<bar+25> mov $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

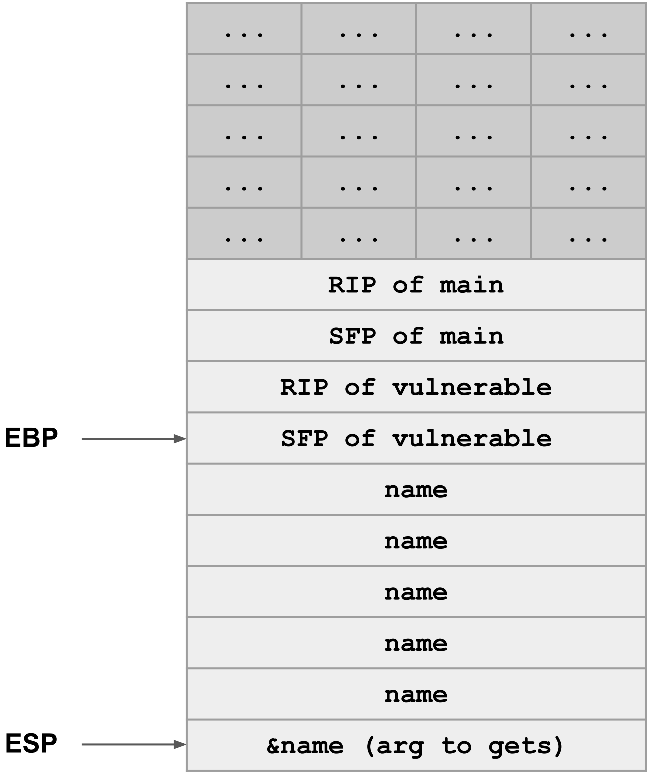
Exploit:

```
'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 → call gets

```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %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)

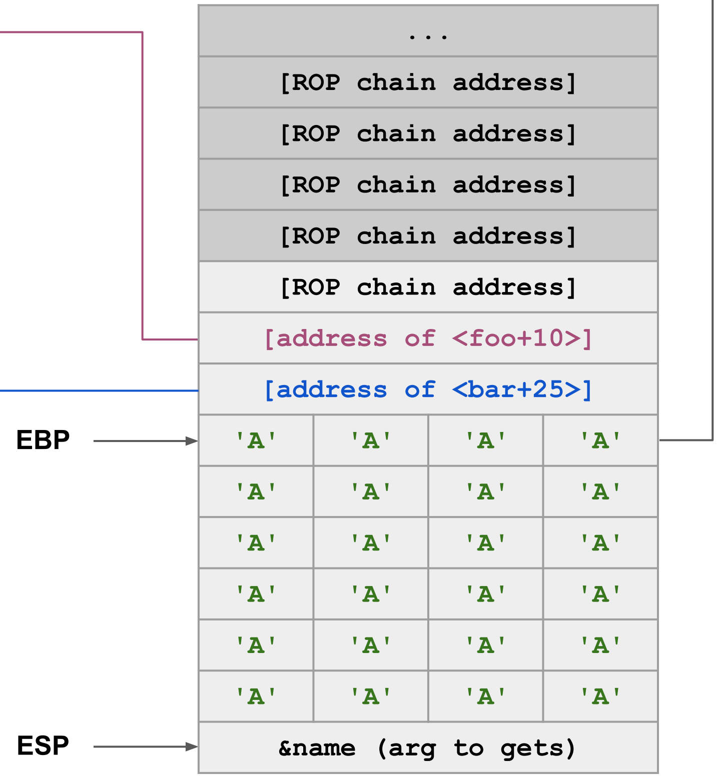
# Subverting Non-Executable Pages: ROP

Exploit:

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

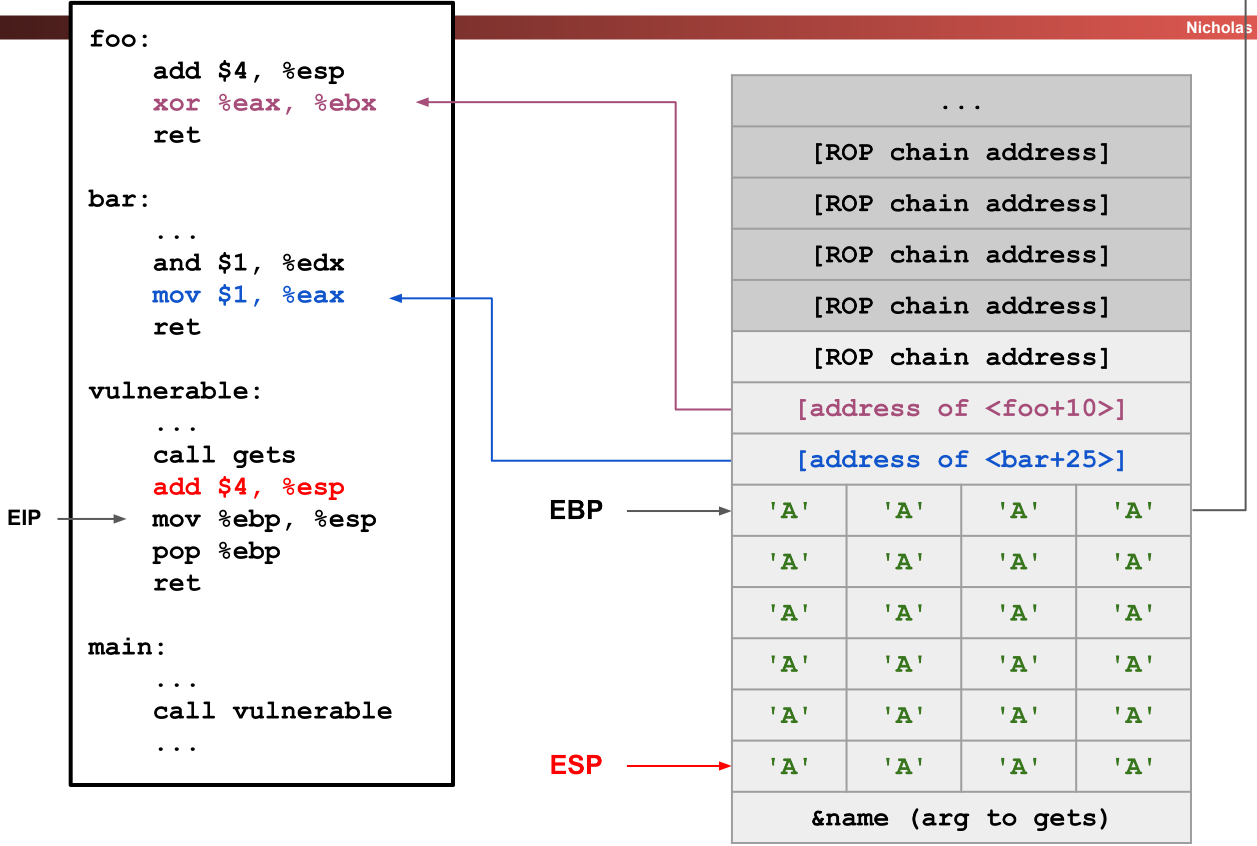
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

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

```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %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:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

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

...

[ROP chain address]

[ROP chain address]

[ROP chain address]

[ROP chain address]

[ROP chain address]

[address of <foo+10>]

[address of <bar+25>]

'A' 'A' 'A' 'A'

'A' 'A' 'A' 'A'

'A' 'A' 'A' 'A'

'A' 'A' 'A' 'A'

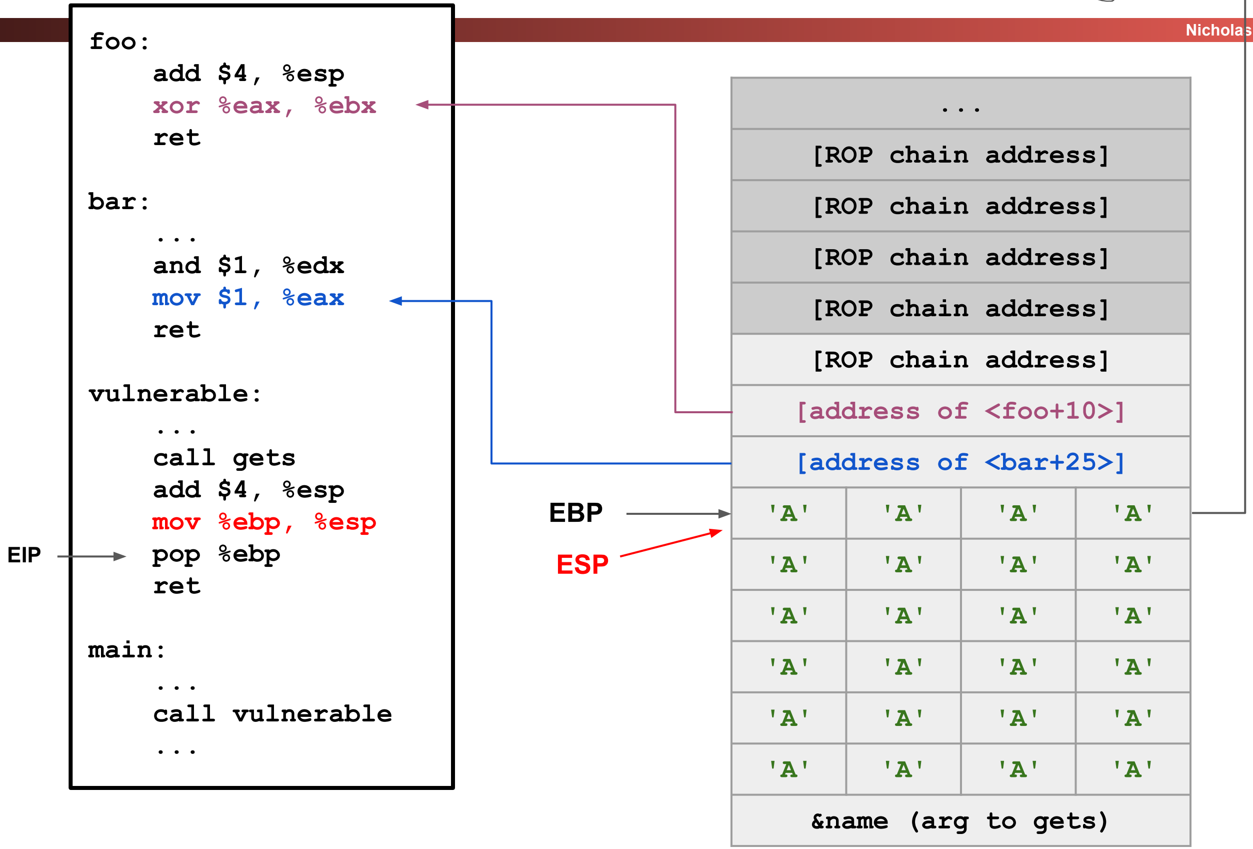
'A' 'A' 'A' 'A'

'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;
}
```

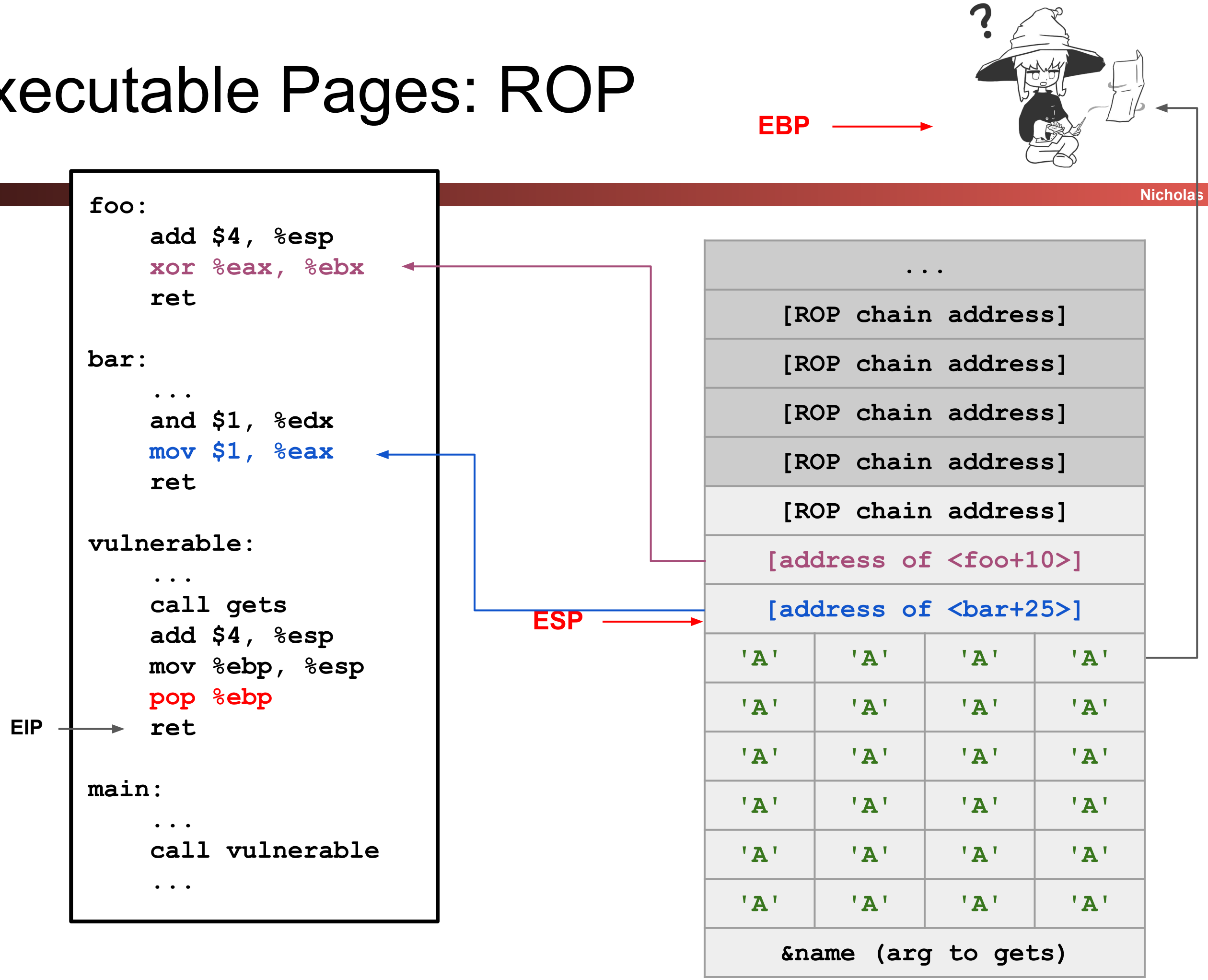


```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

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

```
...
[ROP chain address]
[ROP chain address]
[ROP chain address]
[ROP chain address]
[ROP chain address]
[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;
}
```



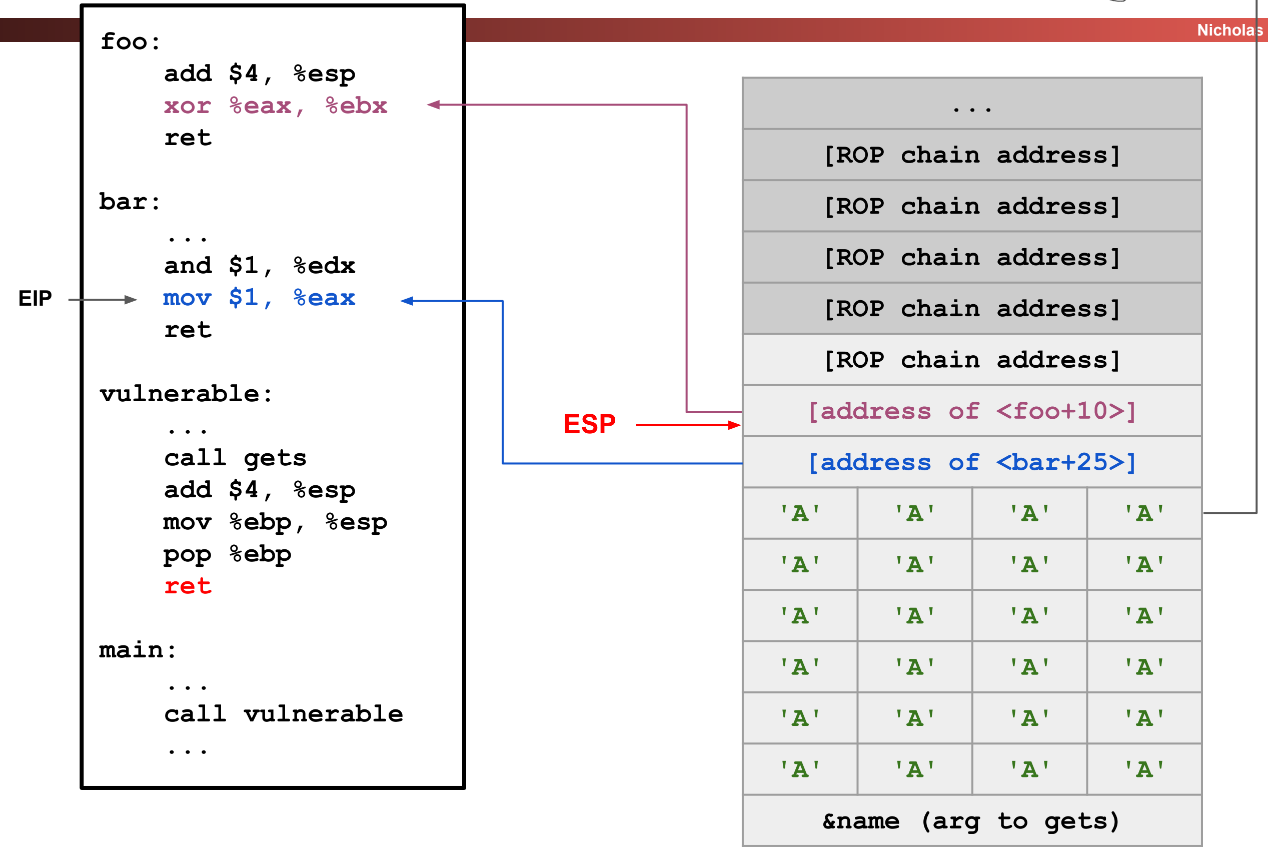
```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

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

| | | | |
|---|---|---|---|
| ... | | | |
| [ROP chain address] | | | |
| [ROP chain address] | | | |
| [ROP chain address] | | | |
| [ROP chain address] | | | |
| [ROP chain address] | | | |
| [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;
}
```

EBP

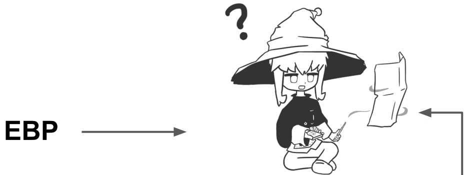


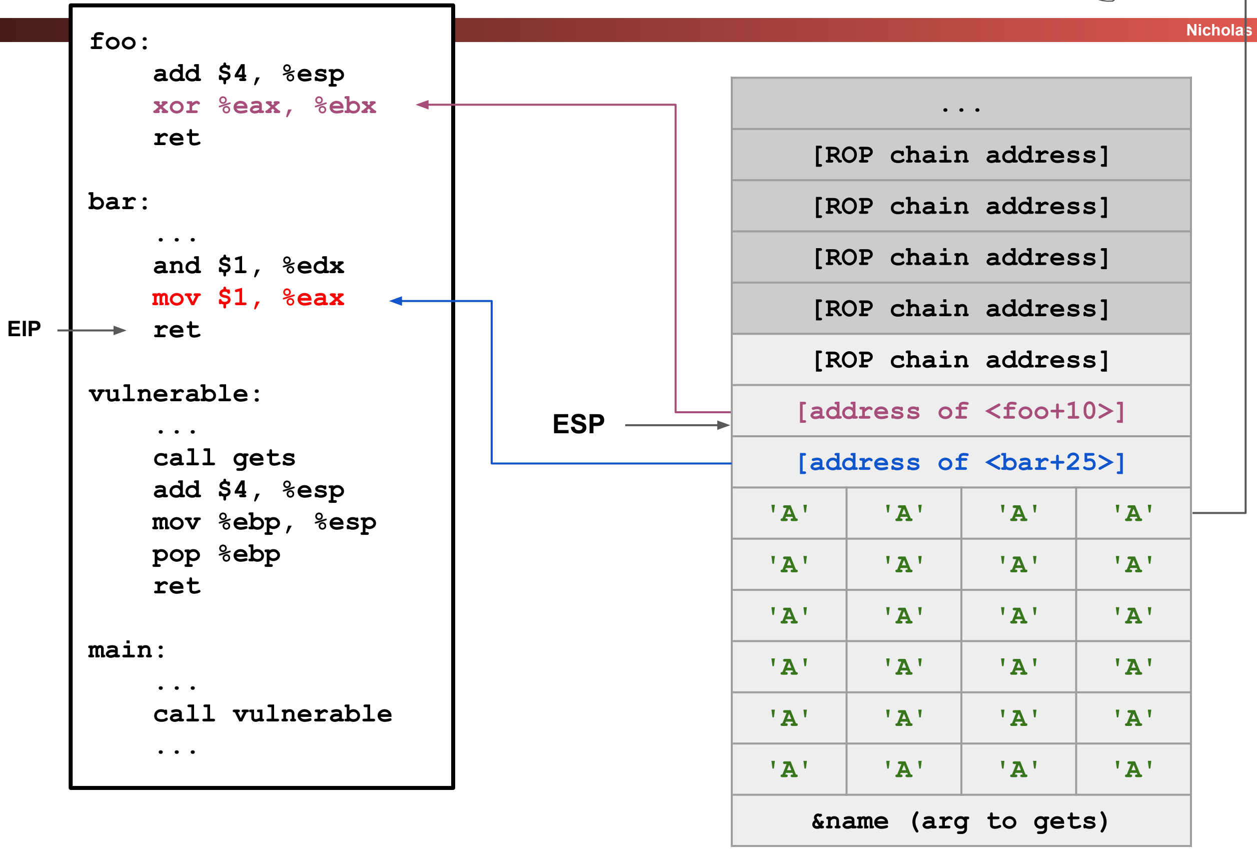
```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

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

```
...
[ROP chain address]
[ROP chain address]
[ROP chain address]
[ROP chain address]
[ROP chain address]
[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;
}
```





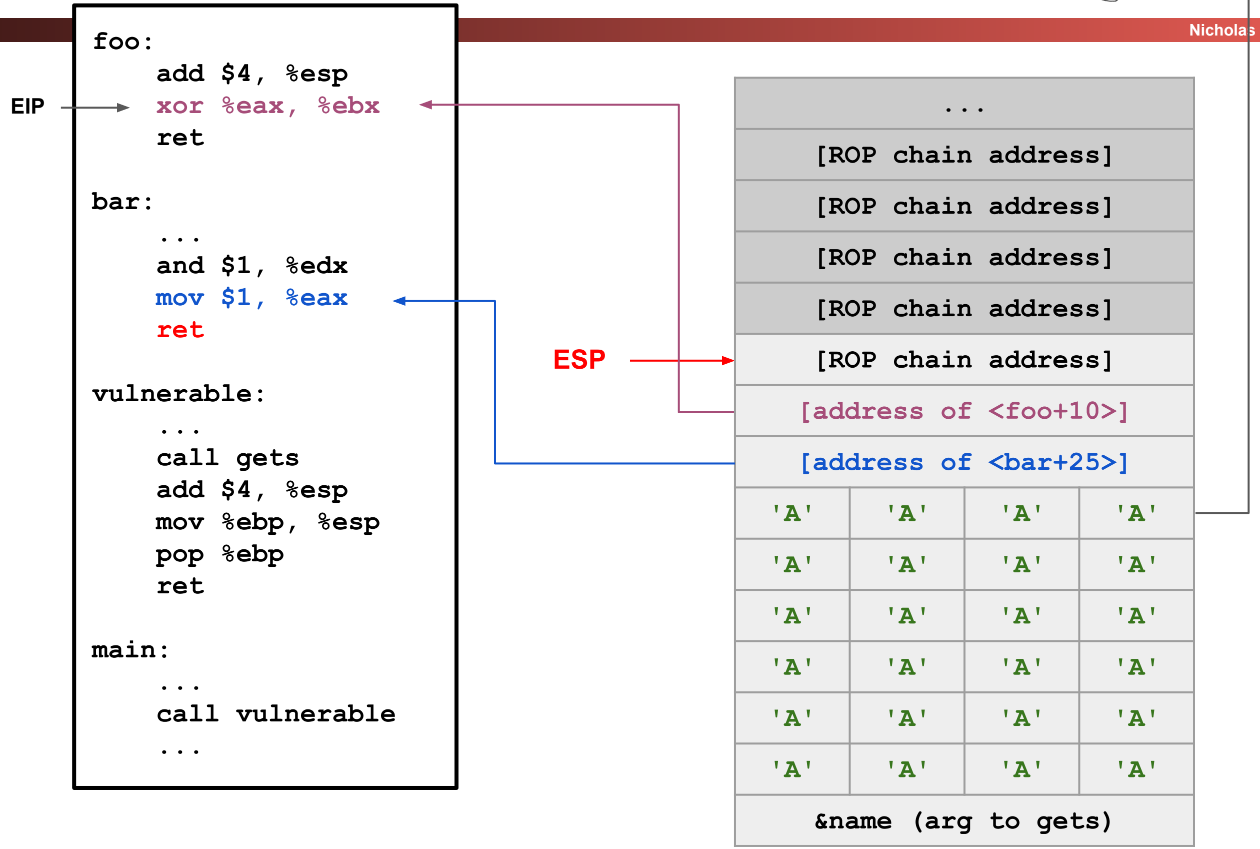
```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

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

| | | | |
|---|---|---|---|
| ... | | | |
| [ROP chain address] | | | |
| [ROP chain address] | | | |
| [ROP chain address] | | | |
| [ROP chain address] | | | |
| [ROP chain address] | | | |
| [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

EBP

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

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



EIP

```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

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

ESP

```
...
[ROP chain address]
[ROP chain address]
[ROP chain address]
[ROP chain address]
[ROP chain address]
[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

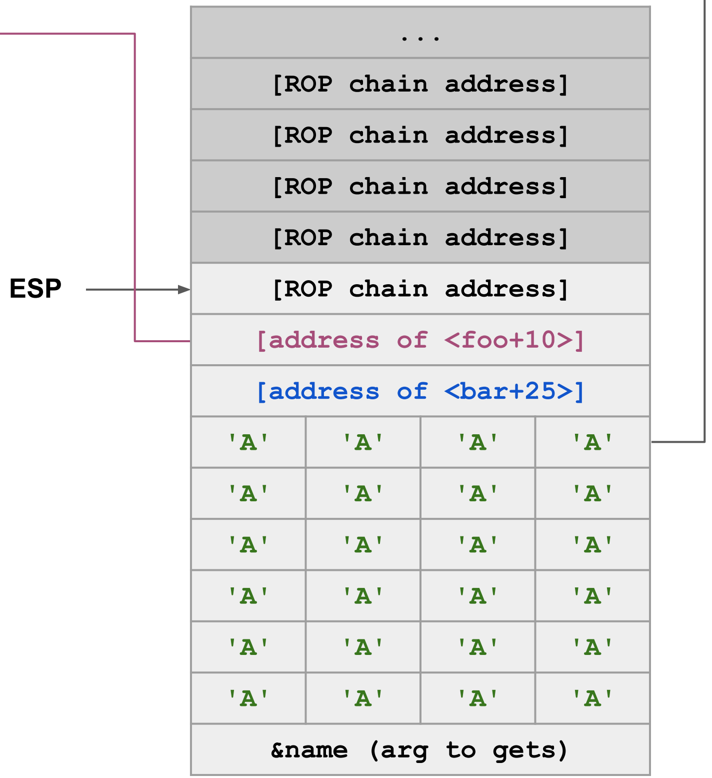
EBP

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;
}
```

EIP

```
foo:
    add $4, %esp
    xor %eax, %ebx
    ret

bar:
    ...
    and $1, %edx
    mov $1, %eax
    ret

vulnerable:
    ...
    call gets
    add $4, %esp
    mov %ebp, %esp
    pop %ebp
    ret

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



ESP

```
...
[ROP chain address]
[ROP chain address]
[ROP chain address]
[ROP chain address]
[ROP chain address]
[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

- 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!
  - ROP was a high-tech attack, and now it is easy and commonplace
- 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