Systems and Internet Infrastructure Security

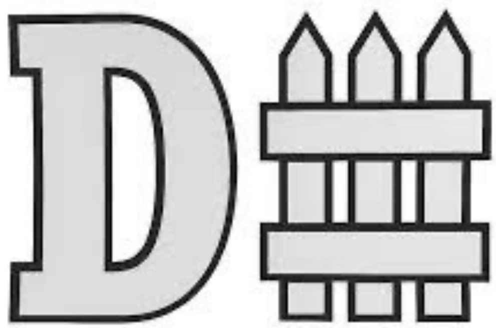
Network and Security Research Center
Department of Computer Science and Engineering
Pennsylvania State University, University Park PA

CMPSC 447
Current Defenses

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

Defenses

- Prevent adversaries from being able to successfully exploit vulnerabilities
  - What enables successful exploitation?



Vulnerability Definition

- A vulnerability is a flaw that is accessible to an adversary who has the ability to exploit that flaw

Vulnerability Defenses

- A vulnerability is a flaw that is accessible to an adversary who has the ability to exploit that flaw
  - So, what is required of an adequate defense to prevent vulnerability exploitation?

Vulnerability Defenses

- A vulnerability is a flaw that is accessible to an adversary who has the ability to exploit that flaw
  - So, what is required of an adequate defense to prevent vulnerability exploitation?
- Prevent one or more of these preconditions
  - Flaw – prevent memory error
  - Access – do not allow adversary input to unsafe operations
  - Exploit – prevent exploit from enabling adversary to achieve their goals
- Think about how each defense relates to these

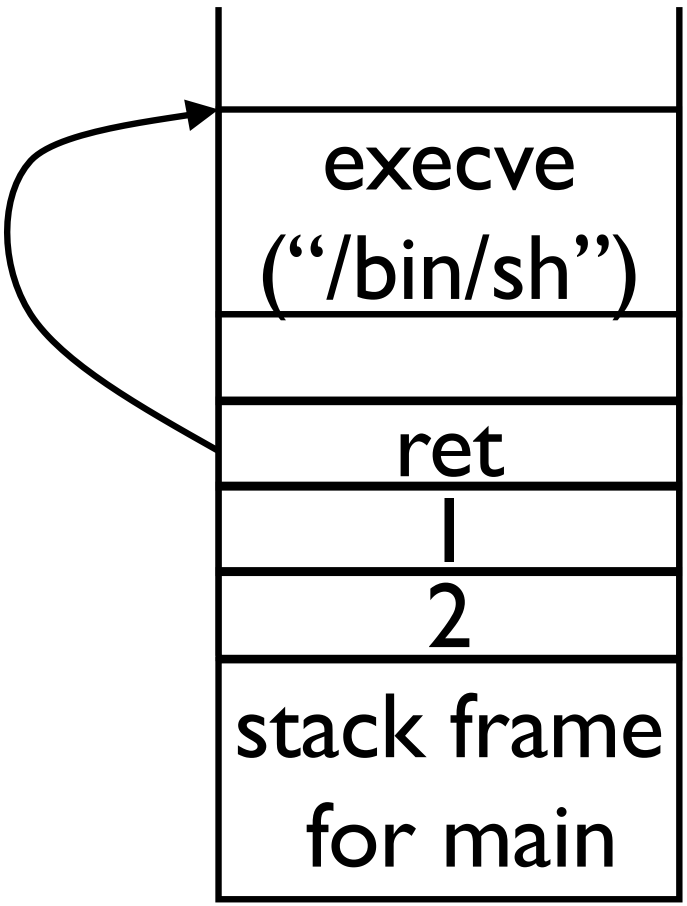
Preventing Buffer Overflows

- How do you prevent buffer overflow attacks?
- Block any of the necessary conditions
  - Check buffer bounds
  - Use a safe function to read input
  - Prevent unauthorized modification of the return address without detection
  - Prevent execution of stack memory
  - Make it impractical for the adversary to find the code she wants to execute, such as "execve"
- Main focus of current defenses is to mitigate spatial errors

Preventing Buffer Overflows

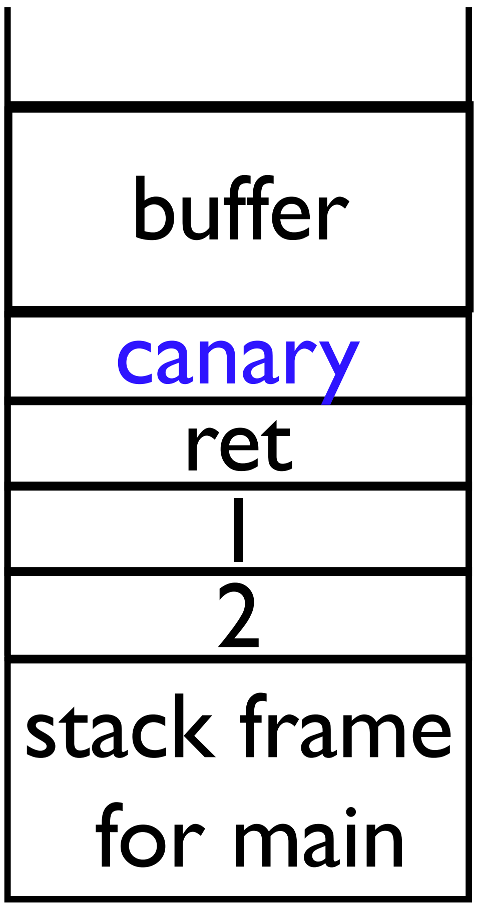
- Block any of the necessary conditions for a vulnerability
  - Check buffer bounds (flaw)
  - Use a safe function to read input (flaw)
  - Prevent unauthorized modification of the return address without detection (exploit)
  - Prevent execution of stack memory (exploit)
  - Make it impractical for the adversary to find the code she wants to execute, such as "execve" (access)
- We spoke about safe programming techniques to reduce the number of flaws
  - Defenses aim to prevent access or exploit options

Buffer Overflow Attack



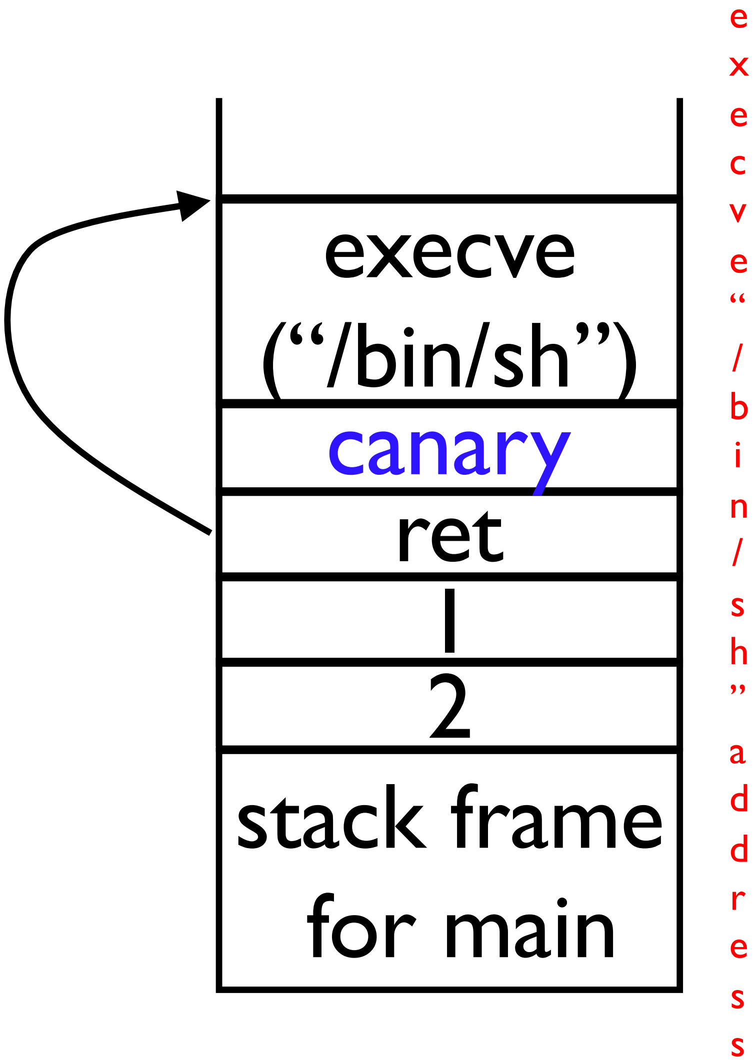
- Remember this exploit
- The adversary's goal is to get execve to run to generate a command shell
- To do this the adversary uses execve from libc – i.e., reuses code that is already there

Stack Canary Defense



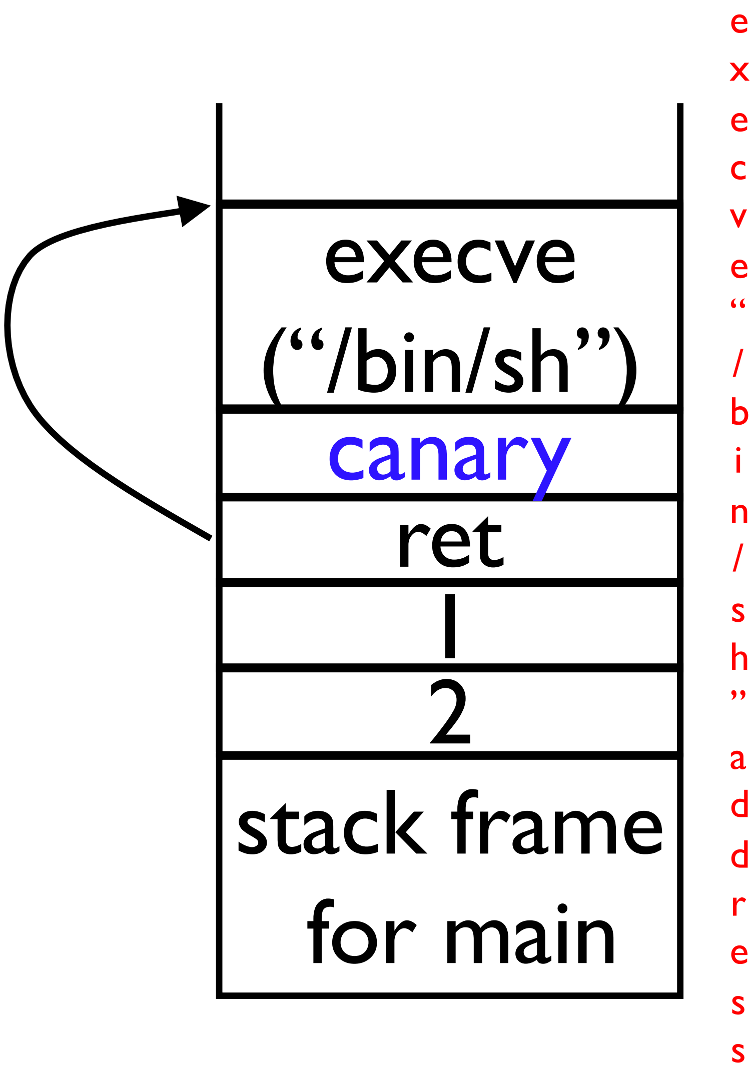
- Place a "canary" value on the stack to detect attempted overwrites of the return address
- Canary value is randomized
- And checked prior to any return
- How does this prevent overflows from exploiting the return address?

Stack Canary Defense



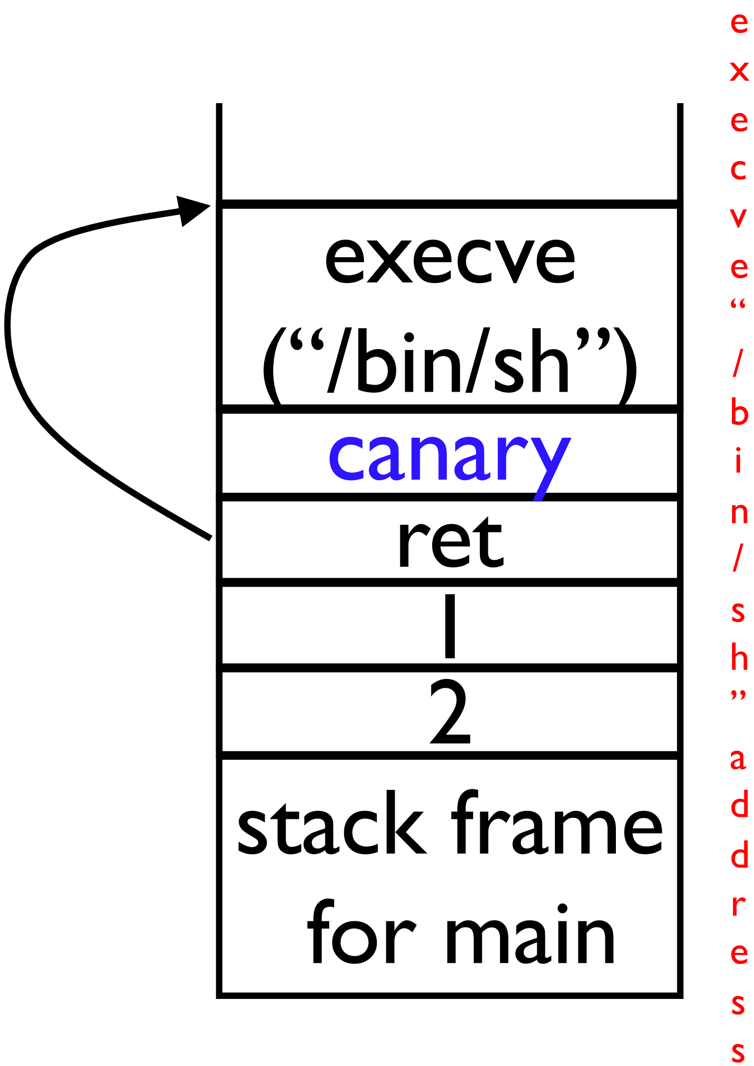
- How does this prevent overflows from exploiting the return address?
- Overflow exploits of the return address from buffer must over overwrite the canary
- But, the canary value is unpredictable – and changes on each run
- So, the check will detect the canary value has changed

Stack Canary Defense



- Limitations of the stack canary defense?

Stack Canary Limitations

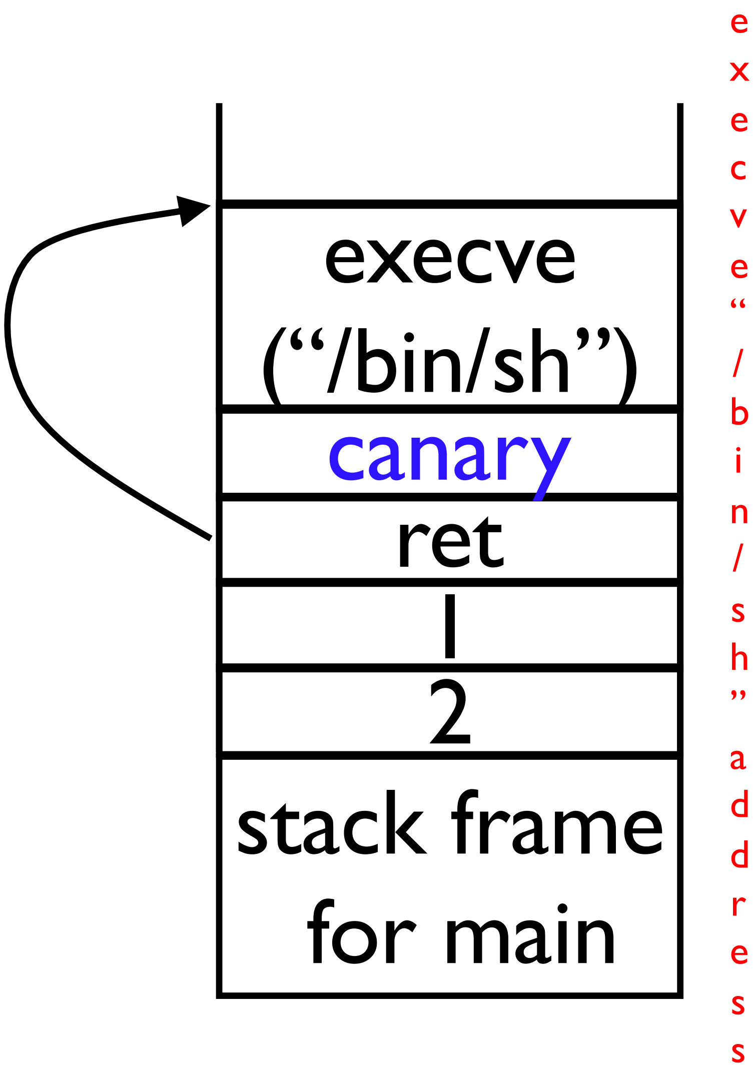


- Limitations of the stack canary defense?
- Must not leak the canary value
- But it is on the stack
  - Readable memory
- What's an attack that may leak the canary?

Buffer Overread/Disclosure

- A buffer overread (disclosure) attack enables an adversary to read memory outside of a region
  - Benign task: Copy from “buffer X” to “buffer Y”
  - Read beyond the memory region of “buffer X”
  - To access other objects’ data
  - And copy into “buffer Y”
- If “buffer X” is on the stack, could possibly read other stack data, including the canary value
  - Once the adversary has read the canary value, they can produce overflow payloads that restore the canary

Stack Canary Limitations



- Limitations of the stack canary defense?
- Only protects the return address

Stack Canary Limitations

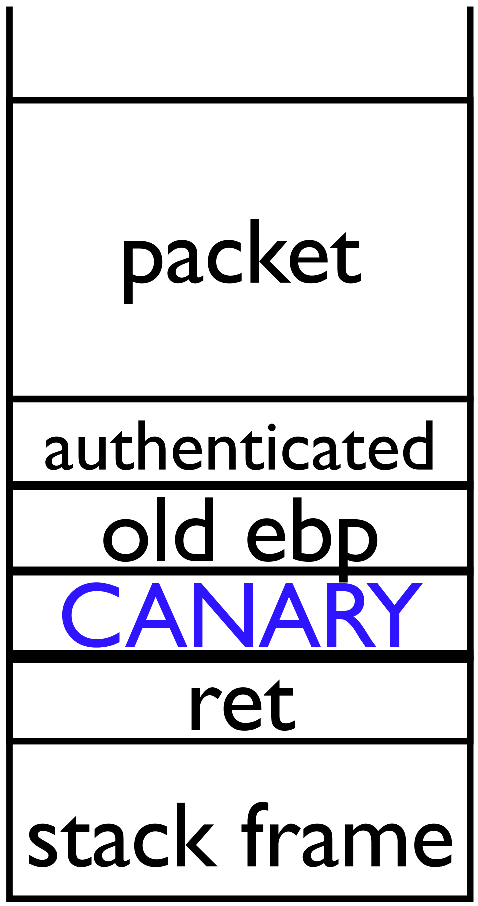
- Obvious limitation: only protects the return address
  - What about other local variables?

```
int authenticated = 0;
char packet[1000];

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
 if (authenticated)
   ProcessPacket(packet);
```

Stack Canary Limitations

- Packet overflows overwrite the authenticated value



Other Approaches

- What is a more straightforward way of checking that the return address hasn't been tampered?

Other Approaches

- What is a more straightforward way of checking that the return address hasn't been tampered?
  - Just check that the value hasn't been tampered
  - Store it somewhere else safe from tampering and check

Shadow Stack

- Method for maintaining return targets for each function call reliably
- On call
  - Push return address on the regular stack
  - Also, push the return address on the shadow stack
- On return
  - Validate the return address on the regular stack with the return address on the shadow stack
- Why might this work? Normal program code cannot modify the shadow stack memory directly

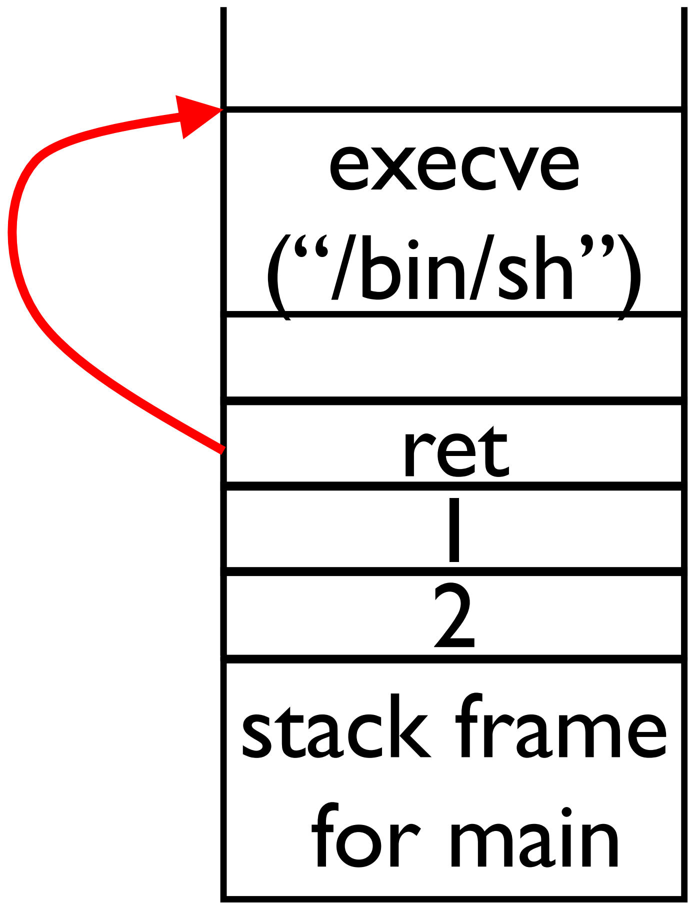
Shadow Stack

- Intel Control-Flow Enforcement Technology (CET)
  - Has been announced
  - Available in 11th generation Intel cores (Tiger Lake)
- Goal is to enforce shadow stack in hardware
  - Throw an exception when a return does not correspond to a call site
- Challenge: Exceptions
  - There are cases where call-return does not match
  - E.g., Tail calls, thread libraries (setjmp, longjmp)

Preventing Buffer Overflows

- Block any of the necessary conditions
  - Check buffer bounds (flaw)
  - Use a safe function to read input (flaw)
  - Prevent unauthorized modification of the return address without detection (exploit)
  - Prevent execution of stack or heap memory (exploit)
  - Make it impractical for the adversary to find the code she wants to execute, such as "execve" (access)
- We spoke about safe programming techniques to reduce the number of flaws
  - Defenses aim to prevent access or exploit options

Buffer Overflow Attack



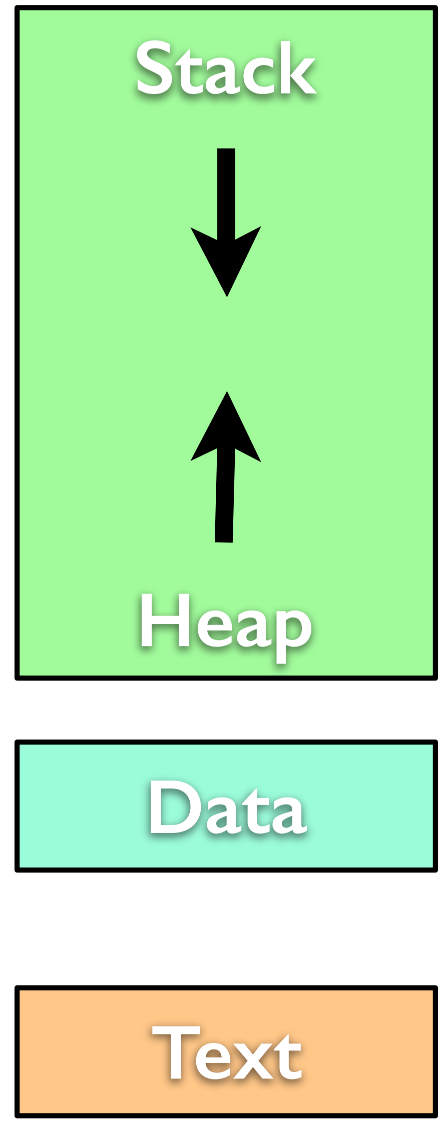
- Suppose there is a buffer overflow flaw
- Inject code on stack
- Set return address to point to the stack
- How to hide the location of the buffer (payload) from the adversary?

Information Hiding

- Prevent access by placing data/code at unpredictable locations
  - Unpredictable == random
- Could randomize the location of all code and data, but would be expensive
- What is a cheap way to randomize a lot of code or data?

Group by Segment

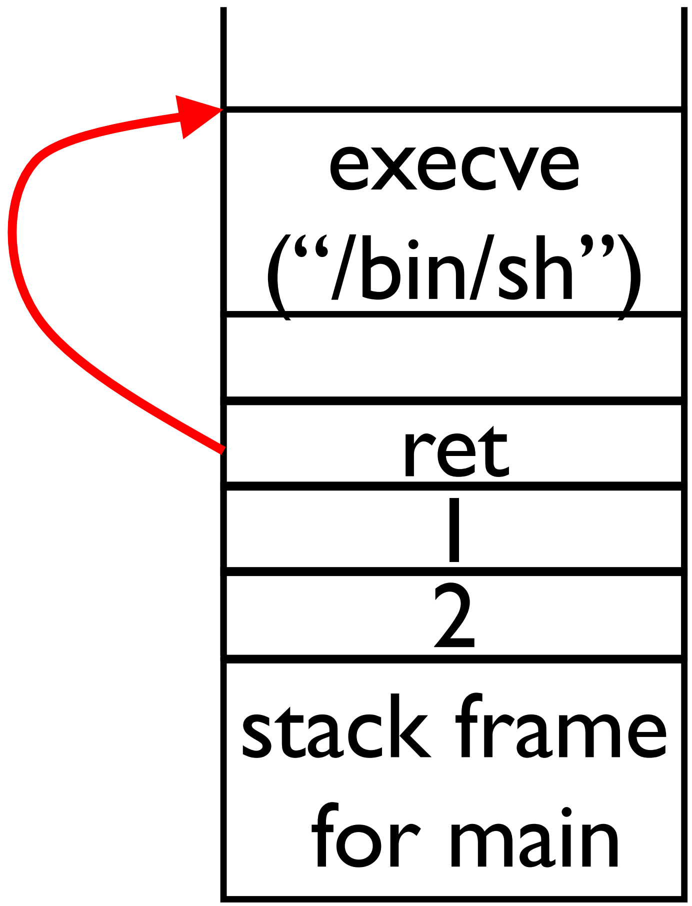
- Move the code and data so that you cannot predict where gadgets will be
  - What is the best way to make unpredictable?
    - Randomize code and data location for each instruction and variable
  - What is the easiest way to make unpredictable?
    - Just move the base address of the segment
    - Called Address Space Layout Randomization



ASLR

- Create a memory segment
  - Heap
  - Stack
  - Code (Library)
- Compute (randomize) the base address
  - High order bits – fixed – segment needs to be placed in the expected relative position
  - Some middle bits – random – this is where ASLR is applied
  - Low order bits – align – must be at least page aligned
- Limits the “entropy” of the randomization
  - Number of possible locations - 2^n where n is entropy in ”bits”

Buffer Overflow Attack



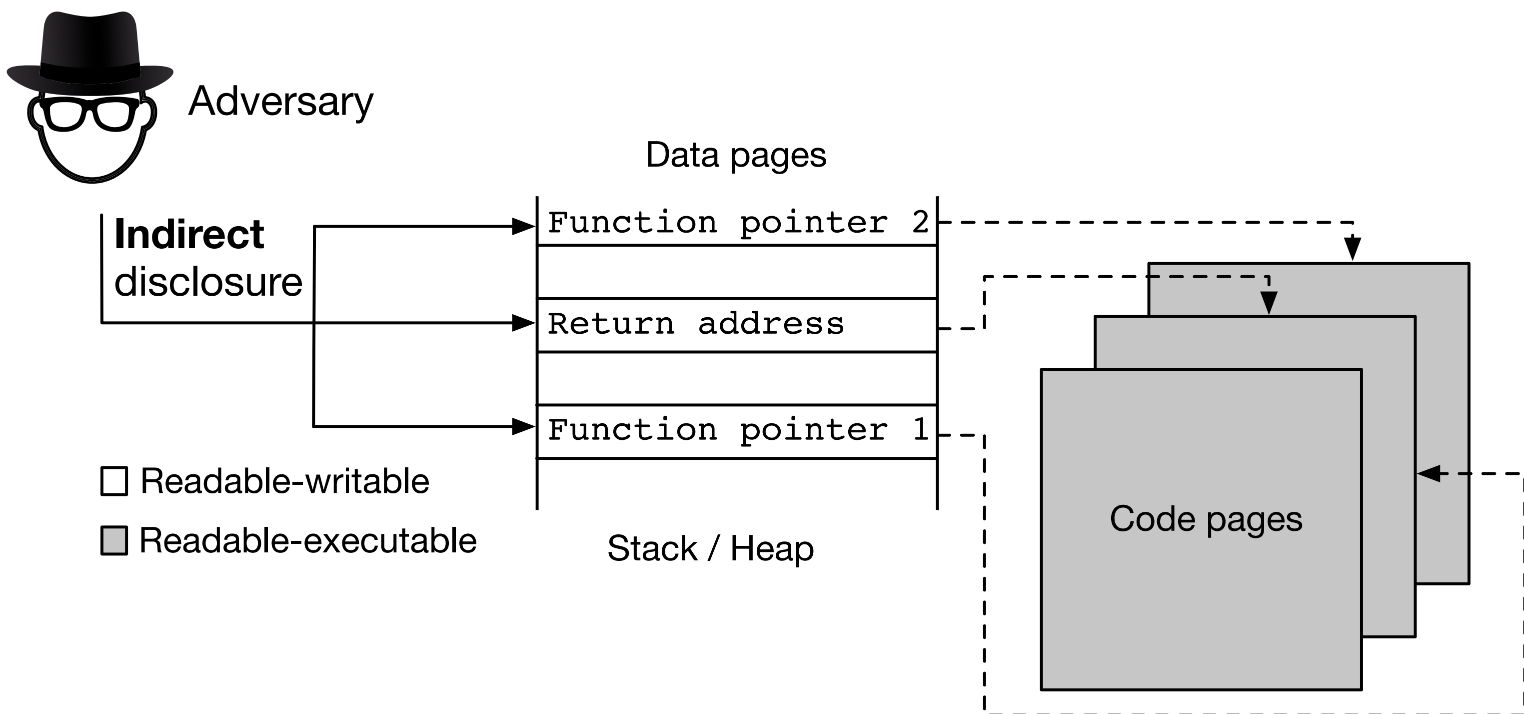
- Suppose there is a buffer overflow flaw
- Inject code on stack
- Set return address to point to the stack
  - With ASLR on the stack segment
- Cannot predict the payload's address

Limitations of ASLR

- What is the risk to ASLR?
  - Memory Disclosure
- Consider a buffer overread
  - E.g., Heartbleed
- Instead of reading a key value
  - What would you read to attack ASLR?

Disclosure Attacks on ASLR

- Adversary harvests pointers stored on the data pages of the application that are necessarily readable

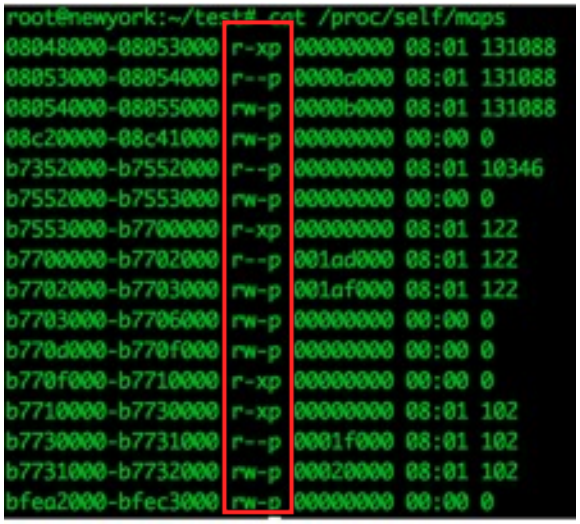


Preventing Buffer Overflows

- Block any of the necessary conditions for a vulnerability
  - Check buffer bounds (flaw)
  - Use a safe function to read input (flaw)
  - Prevent unauthorized modification of the return address without detection (exploit)
  - Prevent execution of stack or heap memory (exploit)
  - Make it impractical for the adversary to find the code she wants to execute, such as "execve" (access)
- We spoke about safe programming techniques to reduce the number of flaws
  - Defenses aim to prevent access or exploit options

DEP … W xor X

- An approach to prevent code injection on the stack is to make the stack non-executable
- Technique is called DEP (Windows) and W xor X (Linux)
- Idea: Each memory region is either writable (like data) or executable (like code), but not both
- Prevents code injection on stack, but not invoking functions directly



How To Use DEP

- Set the program memory regions to be either writable or executable, but not both
  - Writable: ???
  - Executable: ???
  - Of course, some can be read-only and not executable

How To Use DEP

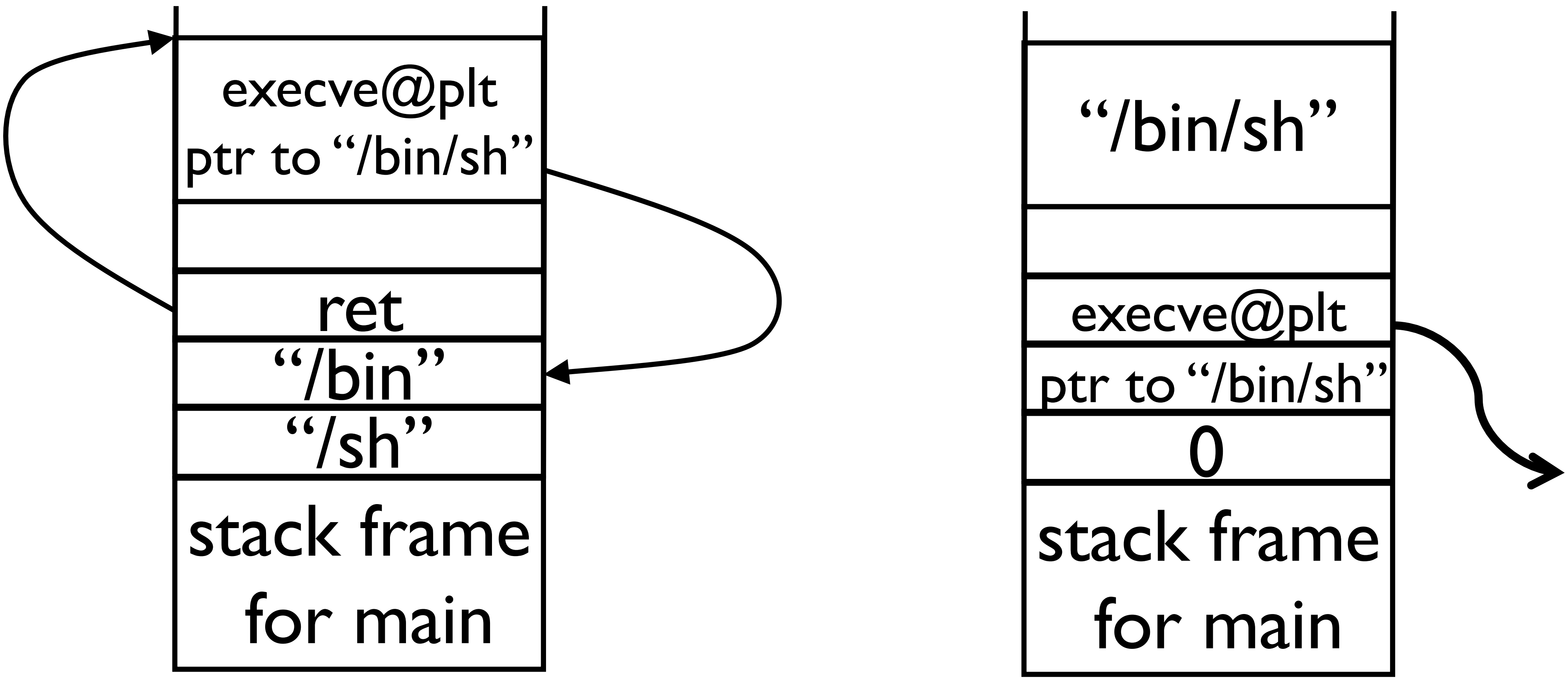
- Set the program memory regions to be either writable or executable, but not both
  - Writable: Stack and heap and global data
  - Executable: Code
  - Of course, some can be read-only and not executable
- Bottom line is that we can remove the execute permission from stack and heap memory pages
  - And prevent writing of code pages
  - To prevent all forms of code-injection attacks

DEP Limitations

- Big limitation: code injection is not necessary to construct adversary-controlled exploit code
  - Attacks that bypass DEP?

Code-Reuse Attacks

- How can we invoke execve without code injection?
  - Use the code directly
- The difference is subtle, but significant



Disable DEP

- How would we use code reuse to disable DEP?
- Goal is to allow execution of writable memory (i.e., change page permissions)
  - There's a system call for that

```
int mprotect(void *addr, size_t len, int prot);
```

  - Sets protection for region of memory starting at address
  - Invoke this system call to allow execution on stack and then start executing from the injected code

Current State of Defenses

PENNSTATE 1855

- Limited
- Protect very little data directly
  - Return addresses (canary or shadow stack)
- Only prevents a subset of exploits
  - Code-reuse attacks still possible with DEP
- Prone to circumvention
  - Disclosures can compromise canary and ASLR defenses
  - Can disable DEP using mprotect
- But, these defenses have modest overhead

Take Away

- Today, we examined defenses that are available by default on current systems
- These defenses aim to prevent vulnerabilities from being exploited
  - Even if the software has flaws
  - By denying the other preconditions of a vulnerability
    - (1) Access to the flaw and (2) Ability to exploit the flaw
- Key goals – low overhead and compatibility
  - Attacks – code injection and return address hijack
  - Limited scope of protection and may be circumvented