SC3010
Computer Security

Lecture 4: Software Security (III)

Outline

- **Safe Programing**
- **Software Testing**
- **Compiler and System Support**

Outline

- **Safe Programing**
- Software Testing
- Compiler and System Support

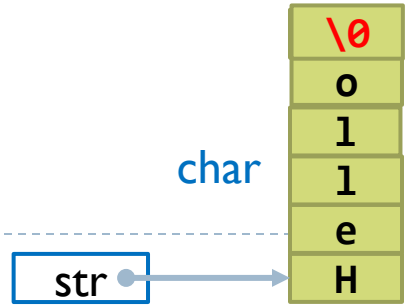
Safe Functions

Root cause: unsafe C lib functions have no range checking

- `strcpy (char *dest, char *src)`
- `strcat (char *dest, char *src)`
- `gets (char *s)`

- Use "safe" versions of libraries:
  - `strncpy (char *dest, char *src, int n)`
    - Copy *n* characters from string src to dest
    - Do not automatically add the NULL value to dest if *n* is less than the length of string src. So it is safer to always add NULL after `strncpy`.
  - `strncat (char *dest, char *src, int n)`
  - `fgets(char *BUF, int N, FILE *FP);`
  - Still need to get the byte count right.

```
char str[6];
strncpy(str, "Hello, World", 5);
str[5] = '\0';
```



Assessment of C Library Functions

Extreme risk

- `gets`

High risk

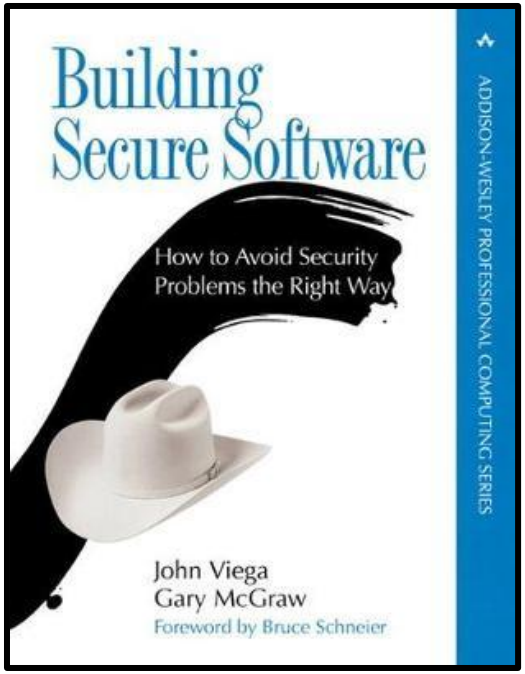
- `strcpy, strcat, sprintf, scanf, sscanf, fscanf, vfscanf, vsscanf, streadd, strecpy, strtrns, realpath, syslog, getenv, getopt, getopt_long, getpass`

Moderate risk

- `getchar, fgetc, getc, read, bcopy`

Low risk

- `fgets, memcpy, snprintf, strccpy, strcadd, strncpy, strncat, vsnprintf`



Safe Libraries

libsafe

- Check some common traditional C functions
  - Examines current stack & frame pointers
  - Denies attempts to write data to stack that overwrite return address or any parameters

glib.h

- Provides Gstring type for dynamically growing null-terminated strings in C

Strsafe.h

- A new set of string-handling functions for C and C++.
- Guarantees null-termination and always takes destination size as argument

SafeStr

- Provides a new, high-level data type for strings, tracks accounting info for strings; Performs many other operations.

Glib

- Resizable & bounded

Apache portable runtime (APR)

- Resizable & bounded

Safe Language (Strong Type)

Ada, Perl, Python, Java, C#, and even Visual Basic

- Have automatic bounds checking, and do not have direct memory access

C-derivatives: Rust (Mozilla 2010)

- Designed to be a "safe, concurrent, practical language", supporting functional and imperative-procedural paradigms
- Does not permit null pointers, dangling pointers, or data races
- Memory and other resources are managed through "Resource Acquisition Is Initialization" (RAII).

Go: type-safe, garbage-collected but C-looking language

- Good concurrency model for taking advantage of multicore machines
- Appropriate for implementing server architectures.

Outline

- Safe Programing
- **Software Testing**
- Compiler and System Support

Manual Code Reviews

Peer review

- Very important before shipping the code in IT companies

Code review checklist

- Wrong use of data:

  variable not initialized, dangling pointer, array index out of bounds, …
- Faults in declarations

  undeclared variable, variable declared twice, …
- Faults in computation

  division by zero, mixed-type expressions, wrong operator priorities, …
- Faults in relational expressions

  incorrect Boolean operator, wrong operator priorities, …
- Faults in control flow

  infinite loops, loops that execute n-1 or n+1 times instead of n, ...

Writing Software Tests

Unit tests

- Test individual components or functions of the software in isolation
- Unit tests should cover all code, including error handling

Regression tests

- Test that new code changes do not negatively affect existing functionality
- Verify that the software continues to function correctly after updates

Integration tests

- Test the interaction between multiple software modules or systems
- Ensure that components work together as expected.

Static Analysis

Analyze the source code or binary before running it (during compilation)

- Explore all possible execution consequences with all possible input
- Approximate all possible states
- Identify issues during development, reducing the cost of fixing vulnerability
- Rely on predefined rules or policies to identify patterns of insecure coding practice

Static analysis tools

- Coverity: https://scan.coverity.com/
- Fortify: https://www.microfocus.com/en-us/cyberres/application-security
- GrammarTech: https://www.grammatech.com/

Limitations

- May produce false positives, requiring manual review
- Cannot detect runtime issues, e.g., logical errors, dynamic environment-specific flaws

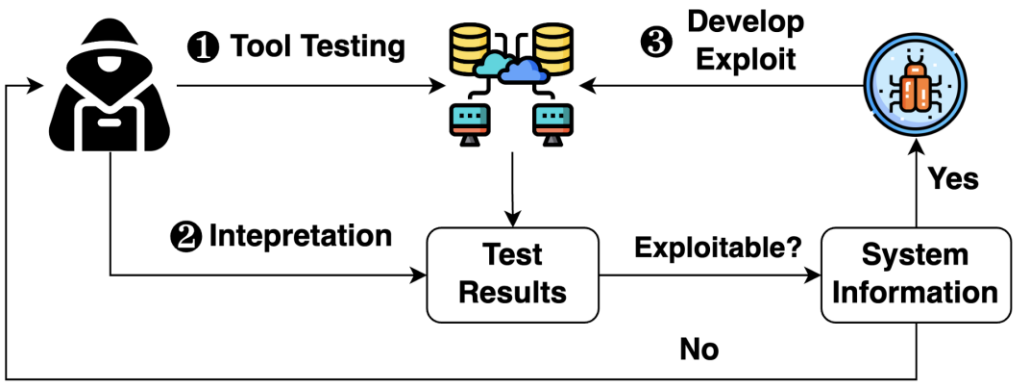
Dynamic Analysis: Penetration Testing

A proactive security assessment method

- Simulate attacks on a system to identify its weakness that is exploitable.
- Identify vulnerabilities before attackers do.
- Ensure compliance with security regulations and improve the overall security posture of systems and applications.

General Procedure

1. Test the system with tools
2. Interpret testing results
3. Check Exploitability
   - Develop the exploit, or
   - Go back to step 1



Dynamic Analysis: Fuzzing

An automated and scalable approach to test software at runtime

- Bombard a program with random, corrupted, or unexpected data to identify how it behaves under unexpected conditions.
- Observe the program for crashes, memory issues or unexpected behaviors.
- Examine failures to determine if they represent exploitable vulnerabilities.

A lot of software testing tools based on fuzzing

- AFL: https://github.com/google/AFL
- FOT: https://sites.google.com/view/fot-the-fuzzer
- Peach: https://wiki.mozilla.org/Security/Fuzzing/Peach

Limitations

- Limited code coverage.
- Require expert analysis to assess whether system crashes are exploitable
- May miss logic flaws that do not result in crashes.

Different Types of Fuzzing Techniques

Mutation-based:

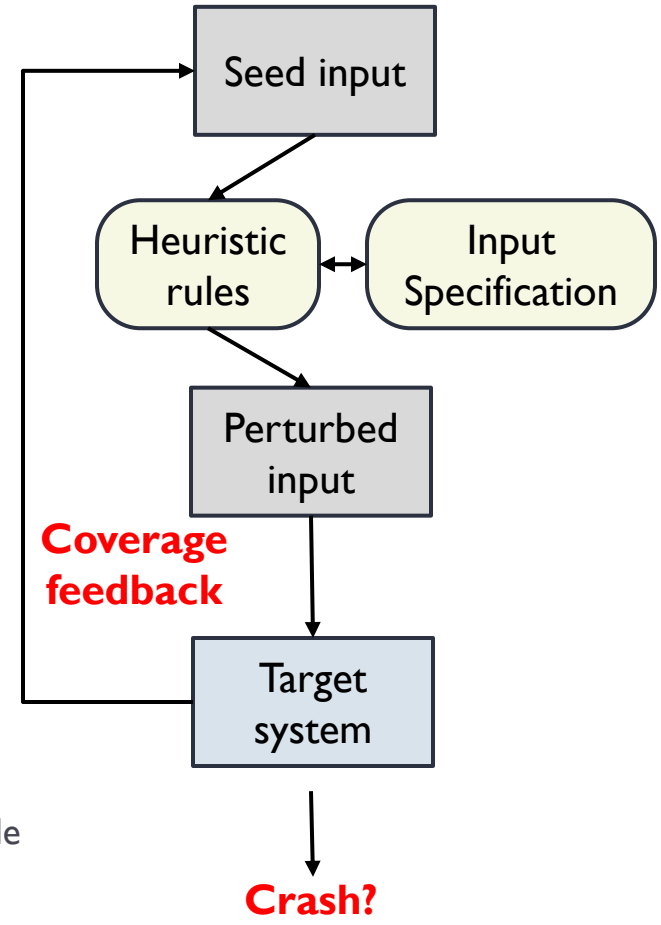
- Collect a corpus of inputs that explores as many states as possible
- Perturb inputs randomly, possibly guided by heuristics, e.g., bit flips, integer increments, substitute with small, large or negative integers.
- Simple to set up. Can be used for off-the-shelf software.

Generation-based

- Convert a specification of input format into a generative procedure
- Generate test cases according to procedure with perturbations
- Get higher coverage by leveraging knowledge of the input format
- Requires lots of effort to set up, and is domain-specific;

Coverage-guided

- Using traditional fuzzing strategies to create new test cases.
- Test the program and measure the code coverage.
- Using code coverage as a feedback to craft input for uncovered code
- Good at finding new states, and combine well with other solutions;



Outline

- Safe Programing
- Vulnerability Detection
- **Compiler and System Support**

Recall: Steps of Stack Smashing Attack

1. Find a buffer overflow vulnerability in the program
2. Inject shellcode into a known memory address
3. Exploit the buffer overflow vulnerability to overwrite EIP with the shellcode address.
4. Return from the vulnerable function.
5. Start to execute the shellcode.

Key insight of defense:

- Make some critical steps more difficult or even impossible to achieve.
- The attacker can only crash the system, but not hijack the control flow to execute arbitrary code.
- This is possibly denial-of-service attacks. Availability is not the main consideration of our threat model. Integrity is more important.

Recall: Steps of Stack Smashing Attack

1. Find a buffer overflow vulnerability in the program
2. Inject shellcode into a known memory address
3. Exploit the buffer overflow vulnerability to overwrite EIP with the shellcode address.
4. Return from the vulnerable function.
5. Start to execute the shellcode.

Solution:

- Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR)

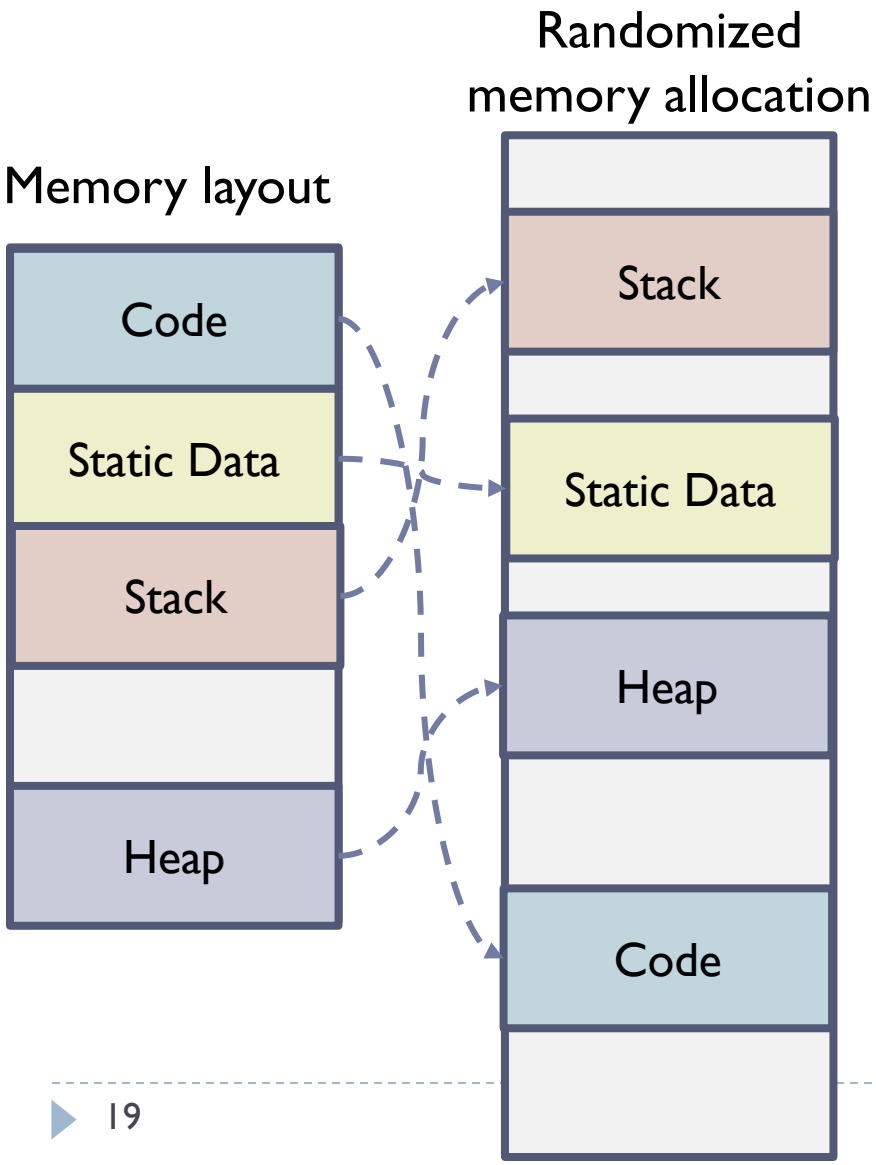
Key idea

- Put the segment of each memory region (data, code, stack, heap, etc.) in a random location every time the program is launched.
- Make it harder for the attacker to get the address of his malicious function
- Within each segment, relative addresses are the same
- Program remain correct if base pointers of these regions are set up correctly
- No performance overhead

ASLR is practical and widely deployed in mainstream systems

- Linux kernel since 2.6.12 (2005+)
- Android 4.0+
- iOS 4.3+ ; OS X 10.5+
- Microsoft since Windows Vista (2007)

ASLR Example




```c
#include <stdio.h>
#include <stdlib.h>
void main() {
    char x[12];
    char *y = malloc(sizeof(char)*12);
    printf("Address of buffer x (on stack):
0x%x\n", x);
    printf("Address of buffer y (on heap) :
0x%x\n", y);
}
```

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

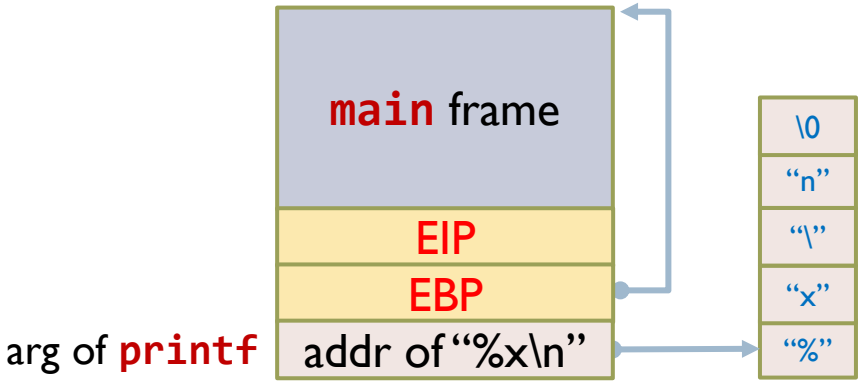
```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

Insecurity of ASLR

Attacker gets the base address of stack first. As the relative addresses within the stack are normally fixed, the attacker can compute the addresses of any data in the stack.

- The attacker can use brute-force technique to guess the base address.
- Format string vulnerability allows the attacker to print out base pointer from the stack (%x).

```
int main(void){

    printf("%x\n");
    return 0;
}
```



Recall: Steps of Stack Smashing Attack

1. Find a buffer overflow vulnerability in the program
2. Inject shellcode into a known memory address
3. Exploit the buffer overflow vulnerability to overwrite EIP with the shellcode address.
4. Return from the vulnerable function.
5. Start to execute the shellcode.

Solutions:

- StackGuard
- Shadow Stack
- StackShield
- PointGuard
- Pointer Authentication

StackGuard

Key insight

- It is difficult for attackers to only modify the return address without overwriting the stack memory in front of the return address.

Steps

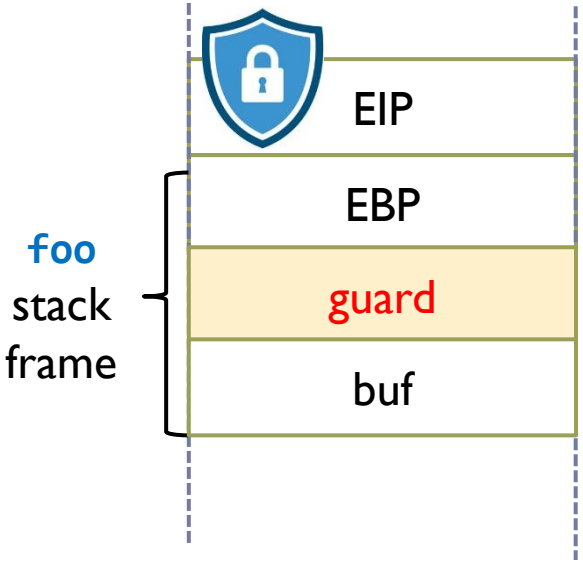
- Embed a canary word next to the return address (EIP) on the stack whenever a function is called.
  - Canary value needs to be random and cannot be guessed by attacker.
- When a stack-buffer overflows into the function return address, the canary has to be overwritten as well
- Every time the function returns, check whether canary value is changed.
- If so, someone is possibly attacking the program with stack-buffer overflows, and the program will be aborted.

First introduced as a set of GCC patches in 1998

How does StackGuard Work

```
void foo(char *s) {


  char buf[16];
  strcpy(buf,s);



}
```

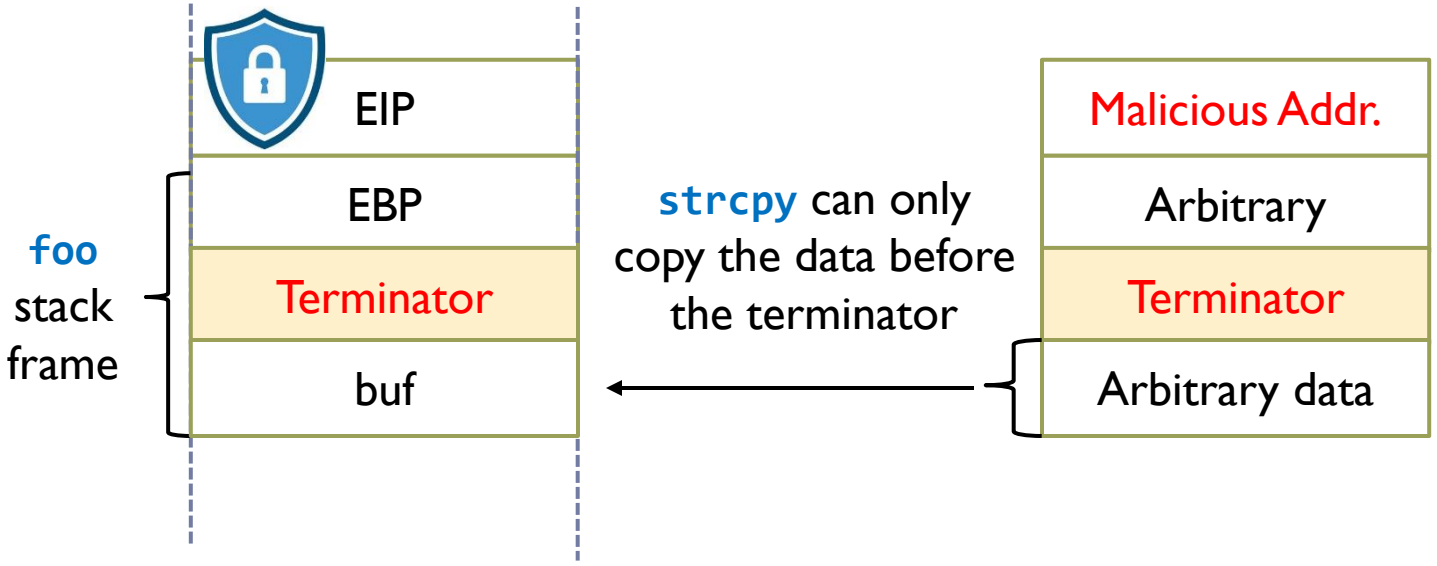
```
int *secret = malloc(size_of(int));
*secret = generateRandomNumber();

void foo(char *s) {
  int guard;
  guard = *secret;

  char buf[16];
  strcpy(buf,s);

  if (guard == *secret)
    return;
  else
    exit(1);
}
```



An Alternative Canary Type

Terminator canary

- Canary = {\0, newline, linefeed, EOF}
- String functions will not copy beyond terminator
- Attacker cannot use string functions to corrupt stack.



Insecurity of StackGuard

Attacker can obtain the canary's value, which will be used to overwrite the canary in the stack without changing the value.

- Format string vulnerability allows the attacker to print out values in the stack (%x).
- The attacker can use brute-force technique to guess the canary.

Attacker can overwrite the return address in the stack without touching the canary.

- Format string vulnerability allows the attacker to write to any location in memory, not need to be consecutive with the buffer (%n).
- Heap overflows do not overwrite a stack canary.

Shadow Stack

Keep a copy of the stack in memory

- On function call: push the return address (EIP) to the shadow stack.
- On function return: check that top of the shadow stack is equal to the return address (EIP) on the stack.
- If there is difference, then attack happens and the program will be terminated.

Shadow stack requires the support of hardware

- Intel CET (Control-flow Enforcement Technology):
  - New register SSP: Shadow Stack Pointer
  - Shadow stack pages marked by a new "shadow stack" attribute:
  - only "call" and "ret" can read/write these pages

StackShield

A GNU C compiler extension that protects the return address.

Separate control (return address) from the data.

- On function call: copies away the return address (EIP) to a non-overflowable area.
- On function return: the return address is restored.
- Even if the return address on the stack is altered, it has no effect since the original return address will be copied back before the returned address is used to jump back.

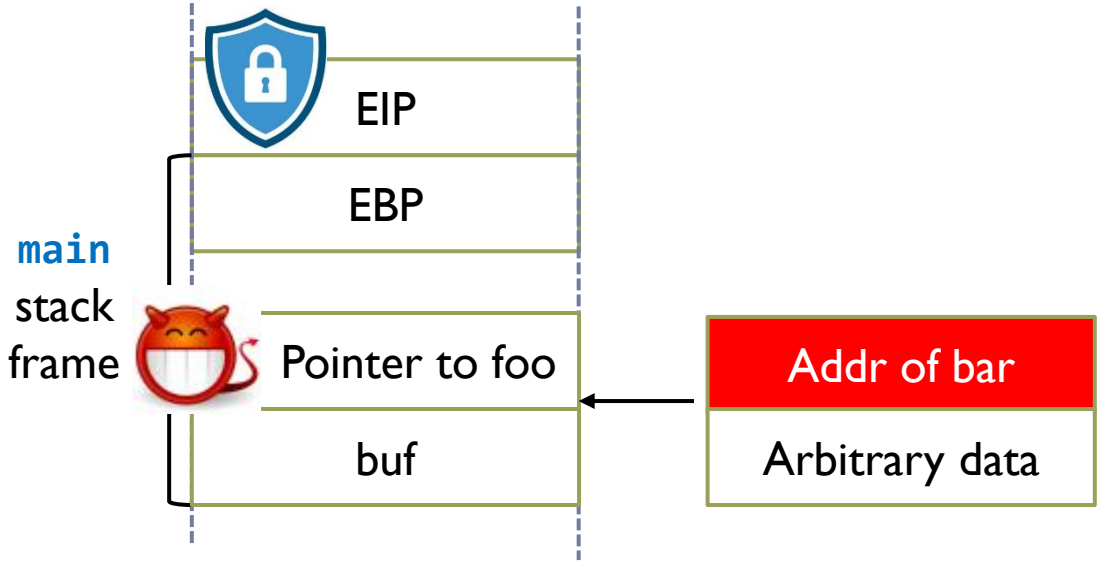
Common Limitations of StackGuard, Shadow Stack, and StackShield

Only protect the return address, but not other important pointers

Hijacking a function pointer

- Even if the attacker cannot overwrite the return address due to the canary, he can overwrite a function pointer.

```
void foo () {...}
void bar () {...}
int main() {
    void (*f) () = &foo;
    char buf [16];
    gets(buf);
    f();
}
```



PointGuard

A compiler-based approach to protect the function pointers from being overwritten

- Encrypt all points while stored in memory, and decrypt them when loaded into CPU registers for use.

Steps

- A secret key is randomly generated for each program when it is launched.
- Pointer encryption: when loading a pointer to memory, encrypt it with the key (typically XOR).
- Pointer decryption: before a pointer is used by CPU, decrypt it with the key (XOR). This ensures the pointer is in its original, unencrypted form only during actual use within CPU, minimizing the window of vulnerability.
- Without knowing the correct key, the attacker cannot overwrite stack data with the encrypted form of malicious function address.

Pointer Authentication

Introduced in ARM architecture to protect function pointers

- Appending a cryptographic signature, known as a Pointer Authentication Code (PAC) to pointers.
- Allowing the CPU to verify the integrity of pointers before they are used.

Steps

- Pointer signing: when a pointer is created or updated, a PAC is generated using a cryptographic hash of the pointer's value and a secret key. This PAC is then embedded into the unused high-order bits of the pointer.
- Pointer verification: before a pointer is used by CPU, the system verifies its integrity by recalculating the PAC and comparing it to the one stored in the pointer. The pointer can be used only when the PAC values match.
- Without knowing the correct key, the attacker cannot generate the correct PAC for his malicious function pointer, and pass the pointer verification.

Recall: Steps of Stack Smashing Attack

1. Find a buffer overflow vulnerability in the program
2. Inject shellcode into a known memory address
3. Exploit the buffer overflow vulnerability to overwrite EIP with the shellcode address.
4. Return from the vulnerable function.
5. Start to execute the shellcode.

Solution:

- Non-Executable Memory

Non-Executable Memory

Key idea

- Attackers inject the malicious code into the memory, and then jump to it.
- We can configure the writable memory region to be non-executable, and thus preventing the malicious code from being executed.
- Windows: Data Execution Prevention (DEP)
- Linux: ExecShield

```
# sysctl -w kernel.exec-shield=1   // Enable ExecShield
# sysctl -w kernel.exec-shield=0   // Disable ExecShield
```

Hardware support

- AMD64 (**NX-bit**), Intel x86 (**XD-bit**), ARM (**XN-bit**)
- Each Page Table Entry (PTE) has an attribute to control if the page is executable

Insecurity of Non-Executable Memory

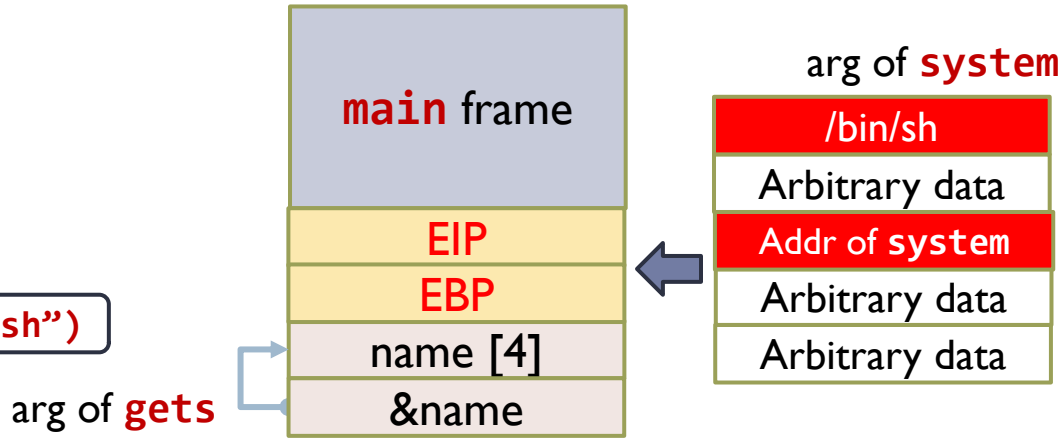
Non-Executable Memory protection does not work when the attacker does not inject malicious code, but just using existing code

Return-to-lib attack:

- Replace the return address with the address of an existing function in the standard C library (libc) or common operating system function.

```
int system(char *command);

void vulnerable(void){
    char name [4];
    gets(name);
}

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

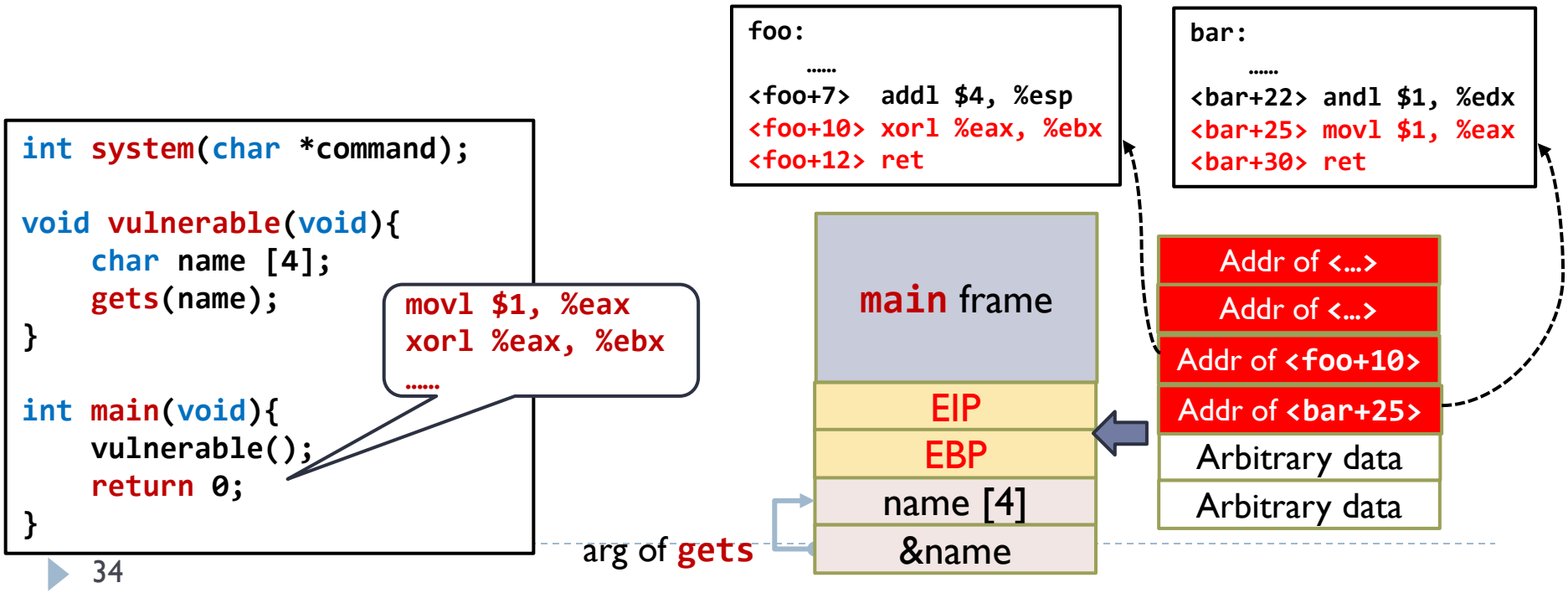
system ("/bin/sh")



Insecurity of Non-Executable Memory

Return-Oriented Programming (ROP):

- Construct the malicious code by chaining pieces of existing code (gadget) from different programs.
- Gadget: a small set of assembly instructions that already exist in the system. It usually end with a return instruction (`ret`), which pops the bottom of the stack as the next instruction.



Limitations of Non-Executable Memory

Two types of executing programs

- Compile a program to the binary code, and then execute it on a machine (C, C++)
- Use an interpreter to interpret the source code and then execute it (Python)

Just-in-Time (JIT) compilation

- Compile heavily-used ("hot") parts of the program (e.g., methods being executed several times), while interpret the rest parts.
- Exploit runtime profiling to perform more targeted optimizations than compilers targeting native code directly

This requires executable heap

- Conflict with the Non-executable Memory protection