Introduction to Information Security
14-741/18-631 Fall 2021
Unit 3: Lecture 4:
Software Vulnerabilities Defenses

Hanan Hibshi hhibshi@andrew

Defenses and Countermeasures I

- **DO NOT use C/C++**
  - Legacy code
  - Practical ???
- **Secure Coding**
  - Avoid risky programming constructs
    - Use `fgets` instead of gets
    - Use `strn`- APIs instead of `str`- APIs
    - Use `snprintf` instead of `sprintf` and `vsprintf`
    - `scanf` & `printf`: use format strings
  - Never assume anything about inputs
    - Do not assume it is coming from a trusted source

Defenses and Countermeasures II

- **Non executable buffers**
  - Prevents injected code from being executed
  - Can affect OS optimizations
- **Array bounds checking**
  - Compiler Improvements
  - Run-time Memory Access Checking (e.g., IBM Purify)
  - Static analysis (e.g. using model checking tools)

Defenses and Countermeasures III

- **StackGuard**
  - Usually referred to as a “stack canary”

Defenses and Countermeasures III

- **StackGuard**
  - Places a canary word immediately after the return address

Defenses and Countermeasures III

- **StackGuard**
  - Places a canary word immediately after the return address
  - Detect change in canary value
    - If value changed, abort!

StackGuard Attacks

- **Fixed Canary**
  - Compiler selects a fixed value
  - Attack?
    - Overwrite canary with its correct value; Overwrite only the return address
- **Random Canary**
  - Random 32-bit value computed at runtime
  - Attack?
    - Format string attack; brute force
- **Terminator Canary**
  - NULL, CR, -1, LF
  - Attacker cannot replace canary since any string function will terminate on receiving one of these values
  - Attack?
- **Secure ???**
  - Canary = (Random Value) XOR (Return Address)
    - Secure only with random canary

StackShield

- **Duplicate Stack**
- **Use return address from the unused stack**
- **3 Forms of Protection**
  - Global Ret Stack
    - Separate stack for return address
      - 256 entries => *protected* function nesting depth of 256
  - Ret Range Check
    - Global variable stores return address of current function
    - Comparison before returning
    - Can detect attack, Global Ret Stack cannot
  - Function Pointer Protection
    - Enforces function pointers to only point to text segment
- **Reference**
  - http://www.angelfire.com/sk/stackshield
  - http://www.madchat.fr/coding/c/c.seku/StackguardPaper.pdf

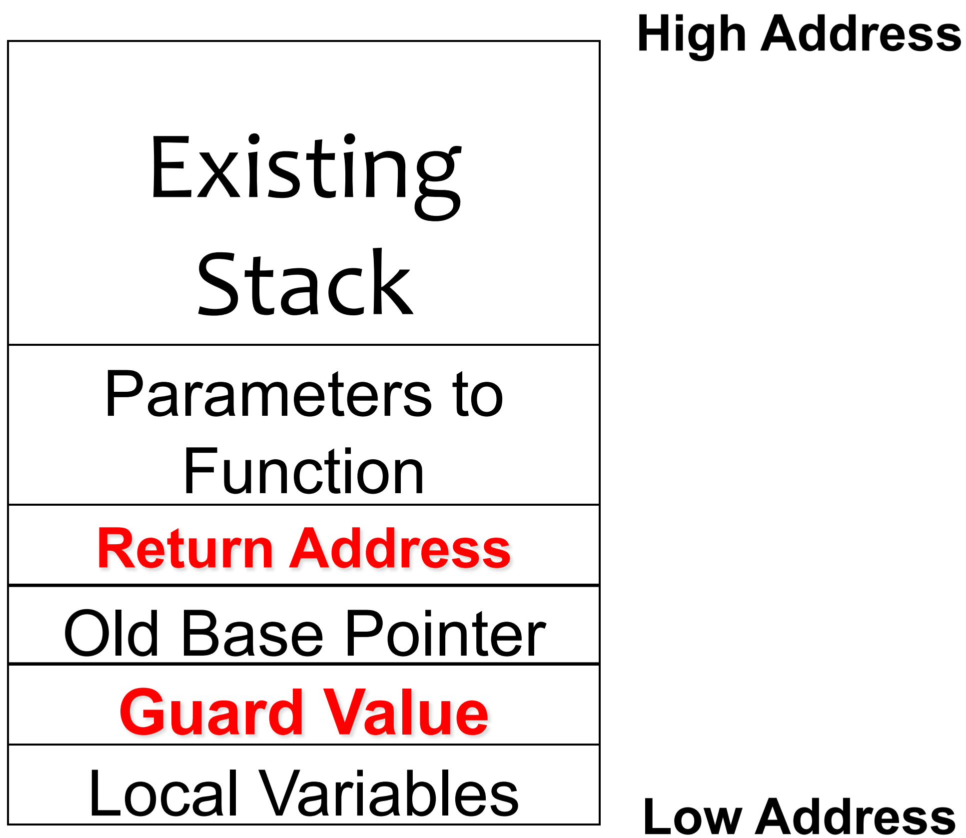
Countermeasure Flaws

- **StackGuard**
  - Attacker could try to keep the canary value but change just the return address
  - Only protect return address on the stack
- **StackShield**
  - Function pointer vulnerabilities
    - Using gadgets!
  - longjmp buffers

ProPolice

- **Uses a *guard (=canary)* value**
- **Rearranges local variables so that char buffers are always at higher addresses than other local variables**
  - Prevents non buffer local variables from being corrupted
- **Change in guard value indicates an attack**
- **Initially developed by IBM for GCC**

ProPolice - II



Libsafe

- **C Library Function Patch**
- **Runtime bounds checking**
  - Wrapper functions - strcpy / strcat / getwd / gets / [vf]scanf / realpath / [v]sprintf
  - Estimates whether a function call will access a buffer beyond a safe boundary
  - "Safe boundary" is defined to protect old base pointer and return address
- **Does not handle non-Return Address attacks**
  - Function pointers
- **Reference**
  - http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails

Libverify

- **Return Address verification like StackGuard at run time**
- **Libverify library**
  - Uses a separate canary stack
  - Function call/return code instrumented to call the canary verification code
- **Advantage:**
  - *No recompilation required*, alters all functions in a process to include special call / return instructions
- **Problem:**
  - Canary stack integrity is not protected

Comparison Parameters

- **Techniques**
  - Overflow buffer all the way to the attack target
  - Overflow the buffer to redirect a pointer to the target
- **Locations**
  - Stack
  - Heap
  - Data segment
- **Attack Targets**
  - Return address
  - Old base pointer
  - Function pointers
  - longjmp buffers

ShortComings

- **Maximum of 40% of attack forms were prevented**
  - 10% detected and halted
- **Program halt => denial of service**
- **Canaries & separate stacks are not protected**
  - Except StackGuard's terminator canary
- **Code recompilation overhead**
- **Limited nesting depth (that is verified)**
- **Protect *only known* attack targets**

Code Reuse Attacks vs. ASLR

- **Canary check happens only at return**
  - Motivates attackers to hijack function pointers
- **NX makes it impossible to inject shellcode into stack**
  - Motivates attackers to reuse existing code

- **Use overflow to setup arguments in stack and overwrite a function pointer to point to, e.g., `system` in the C standard library**
  - "Return-to-libc" attack
  - **What if the base address of each dynamically-loaded library is randomized? (Address Space Layout Randomization)**

Control Flow Integrity

- **Abadi et al. 2005 proposes a comprehensive solution to *all* control flow hijack attacks**
  1. Collect the control flow graph of the target program during compilation or through “binary analysis”
  2. Inserts check before ***every*** control transfer to ensure control flow integrity
     - **Forward edge:** For each call/jmp instruction, store the set of legal destinations in the binary and check at runtime
     - **Backward edge:** Use a duplicated stack to store true return addresses

CFI Example

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

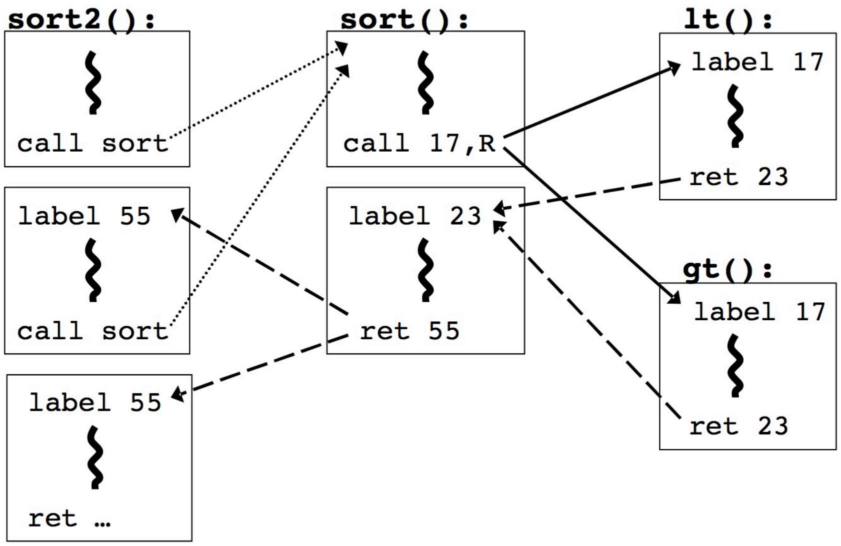


Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.

CFI Instrumentation

- Insert label before the destination
- Add label checking instructions before the computed jump

Source

Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump

Destination

Opcode bytes	Instructions	
8B 44 24 04	mov eax, [esp+4]	; dst
...		

can be instrumented as

Opcode bytes	Instructions	
81 39 78 56 34 12	cmp [ecx], 12345678h	; comp ID & dst
75 13	jne error_label	; if != fail
8D 49 04	lea ecx, [ecx+4]	; skip ID at dst
FF E1	jmp ecx	; jump to dst

Opcode bytes	Instructions	
78 56 34 12	; data 12345678h	; ID
8B 44 24 04	mov eax, [esp+4]	; dst
...		

CFI Requirements

- **UNQ: Each function (equivalent) gets its own unique ID**
  - Cannot clash with another opcode
  - Can be tricky in the presence of dynamic loading
- **NWC: Code segment (“text”) must not be writable**
  - Otherwise, the protection instructions may be overwritten
- **NXD: Data segments must not be executable**
  - Otherwise, attacker may be able to jump to injected code with correct IDs

Current State

- **Stack canary *is* supported by all major compilers on Linux, OS X and BSDs, and Windows**
  - Prevents stack smashing
- **NX-bit *is* (e.g., DEP in Windows)**
  - Non-writeable Code & Non-executable Data
- **Address Space Layout Randomization *is***
  - A “moving target defense” to make calling injected code or code-reuse attack much harder
- **CFI is *not* commonly available**
  - But great progress on Forward CFI in C++ vtable calls
  - NaCL is an example from Google, now we have Web Assembly

Take Away Slide

- **Buffer overflows are still one of the most important source of security vulnerabilities**
- **Programming mistakes and loose access control sink ships**
- **The best thing to do would be to get rid of unsafe programming languages**
  - Not necessarily possible
  - Need to have very good software engineering practices
    - check all bounds
    - shy away from clever optimizations - that's the compiler's job, not yours
  - Compiler tools can help