Lecture 5: Stack-Buffer Overflows II

Summary

In this lecture, we cover some of the major defenses against stack-buffer overflow attacks.

Stack-Buffer Overflow Protection

Reference: This material is not covered in P&P. A good reference is Stallings & Brown, *Computer Security: Principles and Practice,* Section 10.2.

Buffer overflow recap

Attacker must:

1. Overwrite values on stack

2. Execute the code on the stack.

3. Predict location of code in memory.

Modern systems protect against these things.

Stack Protection

Goal is to prevent attacker from overwriting anything important (esp. return address)

Random Canaries

Random word on stack, typically below return address.

Additional code checks for changes in “canary” value

Example: GCC stack protector; controlled through command line argument

`-fstack-protector`	Protect some vulnerable functions
`-fstack-protector-all`	Protect all functions
`-fstack-no-stack-protector`	Disable stack protection

Function Arguments	–
Return Address	`0xbffff498`
Saved Frame Pointer	–
Saved Registers	–
Random Canary	`0xa18a6f6c`

⋮

	`0x00000000`
	`0x00000000`
	`0x00000000`
Start of `char check[64]`	`0x00000000`

Location of a StackGuard Random Canary

Some weaknesses with random canaries

Local variables *may* not be protected

Function arguments *may* not be protected, at least within the vulnerable function

Attacker may be able to retrieve or guess random canary value (low entropy)

Some bits of canary may be derived from “guessable” sources

Stack Execution Protection

Prevent execution of code on the stack

Supported by CPU (some alternatives when there is no CPU support)

On Linux, stack is non-executable by default; controlled by one bit in the ELF executable file header

- `-zexecstack` linker option to enable execution of stack

Some weaknesses with stack execution protection

- CPU may not support it (older CPUs) or it may be disabled in BIOS
- May not be enabled in virtualized environment
- Attackers have developed workarounds
  - Return-to-libc / return-to-system
    - Attacker knows address of library function, e.g. execve()
    - Attacker overwrites return address with library function address
    - Attacker “fixes up” stack to that it (a) has the structure the library function expects, and (2) provides “useful” arguments to the library function
    - Does all this with a buffer overflow!

Address Space Layout Randomization

Goal is to defeat “classic” buffer overflow as well as return-to-libc

- Randomly change the location of library/system functions every time the system is booted.
- Randomly change stack location each time a program is executed
- Randomly change location of buffers `malloc()`-ed on the heap each time a program is executed

Example: Ubuntu implements ASLR on multiple components

Stack ASLR

Libs/MMAP ASLR

Exec ASLR

Brk ASLR

Virtual Dynamically Liked Shared Object (VDSO) ASLR

Write good code

Exercises

1. Consider exercise 3 from lecture 4. Will StackGuard protect against this vulnerability? Why or why not?
2. When might a programmer *legitimately* want to enable execution of code on the stack? Research this and find (or develop) a specific example.
3. Review some of the recent vulnerability announcements on the National Vulnerability Database (http://web.nvd.nist.gov/view/vuln/search). How many are stack-based buffer overflows? What types of buffer overflows, other than stack-based, do you see? Choose one other type of buffer overflow and research how it works in general.