Chapter 10

Buffer and Stack Overflows

Buffer Overflow

• A very common attack mechanism
  – From 1988 Morris Worm to Code Red, Slammer, Sasser and many others
• Prevention techniques known
• Still a major concern due to
  – Legacy of widely deployed buggy software
  – Continued careless programming techniques
Buffer Overflow Basics

• Caused by programming error
• Allows more data to be stored in a fixed sized buffer than capacity available
  – Buffer can be on stack, heap, global data
• Overwriting adjacent memory locations
  – Corruption of program data
  – Unexpected transfer of control
  – Memory access violation
  – Execution of code chosen by attacker

Buffer Overflow

```c
int main(){
    int buffer[10];
    buffer[20] = 37;
}
```

• Q: What happens when this is executed?
• A: Depending on what resides in memory at location "buffer[20]"
  – Might overwrite user data or code
  – Might overwrite system data or code
  – Or program could work just fine

Simple Buffer Overflow

• Consider boolean flag for authentication
• Buffer overflow could overwrite flag allowing anyone to authenticate!

In some cases, Charlie may not be so lucky as in this example.
A Little Programming Language History

- At machine level all data are array of bytes
  - Interpretation depends on instructions used
- Modern high-level languages have a strong notion of type and valid operations depending on data type
  - Not vulnerable to buffer overflows
  - Does incur overhead, some limits on use
- C and related languages have high-level control structures, but allow direct access to memory
  - Hence are vulnerable to buffer overflow
  - Have a large legacy of widely used, unsafe, and hence vulnerable code

Programs and Processes

- Top of memory = high address
- Data = static variables
- Stack == “scratch paper”
  - Dynamic local variables
  - Parameters to functions
  - Return address
- Heap == dynamic data

Function Calls and Stack Frames
Buffer Overflow Attacks

- To exploit a buffer overflow, an attacker
  - Must identify a buffer overflow vulnerability in some program
    - inspection, tracing execution, fuzzing tools
  - Understand how buffer is stored in memory and determine potential for corruption

Stack Buffer Overflow

- Occurs when targeted buffer is located on stack
  - Used by Morris Worm
  - “Smashing the Stack” paper popularized it
- Have local variables below saved frame pointer and return address
  - Hence overflow of a local buffer can potentially overwrite these key control items
- Attacker overwrites return address with address of desired code
  - Program, system library or loaded in buffer

Simplified Stack Example

```c
void func(int a, int b){
    char buffer[10];
}
void main(){
    func(1, 2);
}
```
What happens if buffer overflows?
- Program “returns” to wrong location
- A crash is likely

Charlie has a better idea
- Code injection
- Charlie can run code of his own choosing
- On your machine

Charlie may not know...
1) Address of evil code
2) Location of \texttt{ret} on stack

Solutions
1) Precede evil code with \texttt{NOP} “landing pad”
2) Insert \texttt{ret} many times
Stack Smashing Summary

- A buffer overflow must exist in the code
- Not all buffer overflows are exploitable
  - Things must align just right
- If exploitable, attacker can inject code
- Trial and error is likely required
  - Smashing the Stack for Fun and Profit, Aleph One
- Stack smashing is “attack of the decade”
  - Regardless of the decade...

Stack Smashing Example

- Program asks for a serial number that the attacker does not know
- Attacker does not have source code
- Attacker does have the executable (exe)

- Program quits on incorrect serial number

Example

- By trial and error, attacker discovers apparent buffer overflow

- Note that 0x41 is “A”
- Looks like ret overwritten by 2 bytes!
Example

• Next, disassemble bo.exe to find

```
00400000 55 08 89 E5 01 33 C9 83 EC 18 83 F9 08 0F 15 7F 40 00 00 00
```

The goal is to exploit buffer overflow to jump to address 0x401034

Example

• Find that, in ASCII, 0x401034 is “@^P4”

```
"@^P4"
```

Byte order is reversed? Why?
X86 processors are “little-endian”

Example

• Reverse the byte order to “4^P@” and…

```
"4^P@"
```

Success! We’ve bypassed serial number check by exploiting a buffer overflow

• What just happened?
  • We overwrite the return address on the stack
Example

- Attacker did **not** require access to the source code
- Only tool used was a disassembler to determine address to jump to
- Find desired address by trial and error?
  - Necessary if attacker does not have exe
  - For example, in a remote attack

Example

- Source code of the buffer overflow

```c
#include <stdio.h>
#include <string.h>

main()
{
    char in[10];
    printf("Enter Serial Number\n");
    scanf("%s", in);
    if(strcmp(in, "S123456", 0))
    {
        printf("Serial number is correct\n");
    }
}
```

More Stack Overflow Variants

- Target program can be:
  - A trusted system utility
  - Network service daemon
  - Commonly used library code, e.g. image
Shellcode

- Code supplied by attacker
  - often saved in buffer being overflowed
  - traditionally transferred control to a shell
- Machine code
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently have automated sites/tools

Shellcode

- Shellcode functions
  - Spawn shell
  - Create listener to launch shell on connect
  - Create reverse connection to attacker
  - Flush firewall rules
  - Break out of chroot environment

Buffer Overflow Defenses

- Buffer overflows are widely exploited
- Large amount of vulnerable code in use
  - Despite cause and countermeasures known
- Two broad defense approaches
  - Compile-time - harden new programs
  - Run-time - handle attacks on existing programs
Compile-Time Defenses: Programming Language

- Use a modern high-level language with strong typing
  - Not vulnerable to buffer overflow
  - Compiler enforces range checks and permissible operations on variables
- Do have cost in resource use
- And restrictions on access to hardware
  - So still need some code in C like languages

---

Compile-Time Defenses: Safe Coding Techniques

- If using potentially unsafe languages eg. C
- Programmer must explicitly write safe code
  - By design with new code
  - After code review of existing code, cf. OpenBSD
- Buffer overflow safety a subset of general safe coding techniques
  - Allow for graceful failure
  - Checking have sufficient space in any buffer

---

Compile-Time Defenses: Language Extension, Safe Libraries

- Proposals for safety extensions to C
  - Performance penalties
  - Must compile programs with special compiler
- Have several safer standard library variants
  - New functions, e.g. `strlcpy()`
  - Safer re-implementation of standard functions as a dynamic library, e.g. `Libsafe`
Compile-Time Defenses: Stack Protection

- Add function entry and exit code to check stack for signs of corruption
- Save/check safe copy of return address
  - e.g. Stackshield, RAD
- Use random canary
  - e.g. Stackguard, Win /GS
  - check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support

Stack Smashing Defenses

- **Canary**
  - Run-time stack check
  - Push canary onto stack
  - Canary value:
    - Constant 0x000aff0d
    - Or may depend on ret
  - low
  - high

Microsoft’s Canary

- Microsoft added buffer security check feature to C++ with /GS compiler flag
- Based on canary (or “security cookie”)

**Q:** What to do when canary dies?

**A:** Check for user-supplied “handler”

- Handler shown to be subject to attack
  - Claims that attacker can specify handler code
  - If so, formerly “safe” buffer overflows become exploitable when /GS is used!
Run-Time Defenses:
Non Executable Address Space
• Use virtual memory support to make some regions of memory non-executable
  – e.g. stack, heap, global data
  – Need h/w support in MMU, for example NX (no execute bit)
    • Long existed on SPARC / Solaris systems
    • Recent on x86 Linux/Unix/Windows systems
• Issues: support for executable stack code
  – need special provisions

Run-Time Defenses:
Address Space Randomization
• Randomize place in memory where code is loaded
• Manipulate location of key data structures
  – stack, heap, global data
  – using random shift for each process
  – have large address range on modern systems
    means wasting some has negligible impact
• Also randomize location of heap buffers
• And location of standard library functions

ASLR
• Makes most buffer overflow attacks probabilistic
  – Windows Vista uses 256 random layouts
    • So about 1/256 chance that a buffer overflow will work
  – Similar thing in MacOS and other OSs
• Attacks that try to de-randomize memory layout do exist
Run-Time Defenses:
Guard Pages

- Place guard pages between critical regions of memory
  - Flagged in MMU as illegal addresses
  - Any access aborts process
- Can even place between stack frames and heap buffers
  - At execution time and space cost

Other Overflow Attacks

- Have a range of other attack variants
  - Stack overflow variants
  - Heap overflow
  - Global data overflow
  - Format string overflow
  - Integer overflow
- More likely to be discovered in future
- Some cannot be prevented except by coding to prevent originally