Chapter 10

Buffer Overflow
Buffer Overflow

● Common attack mechanism
  ○ first wide use by the Morris Worm in 1988

● Prevention techniques known
  ○ NX bit, stack canaries, ASLR

● Still of major concern
  ○ Recent examples: Shellshock, Heartbleed
Buffer Overflow/Buffer Overrun

Definition:

A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.
Buffer Overflow Basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
  - locations may hold other program variables, parameters, or program control flow data
  - buffer could be located on the stack, in the heap, or in the data section of the process
- Consequences:
  - corruption of program data
  - unexpected transfer of control
  - memory access violations
Basic Buffer Overflow Example

```c
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strcmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs
Basic Buffer Overflow Stack Values

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Before gets(str2)</th>
<th>After gets(str2)</th>
<th>Contains Value of</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>bfffb4</td>
<td>34fcffbf</td>
<td>34fcffbf</td>
<td>argv</td>
</tr>
<tr>
<td></td>
<td>4...</td>
<td>3...</td>
<td></td>
</tr>
<tr>
<td>bfffb0</td>
<td>01000000</td>
<td>01000000</td>
<td>argc</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>bfffbec</td>
<td>c6bd0340</td>
<td>c6bd0340</td>
<td>return addr</td>
</tr>
<tr>
<td></td>
<td>...@</td>
<td>...@</td>
<td></td>
</tr>
<tr>
<td>bfffb0e</td>
<td>00fcffbf</td>
<td>00fcffbf</td>
<td>old base ptr</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>bfffb4e</td>
<td>00000000</td>
<td>01000000</td>
<td>valid</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>bfffb00</td>
<td>00540140</td>
<td>00540140</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.d.@</td>
<td>.d.@</td>
<td></td>
</tr>
<tr>
<td>bfffbdc</td>
<td>54001540</td>
<td>45053534</td>
<td>str1[4-7]</td>
</tr>
<tr>
<td></td>
<td>T...@</td>
<td>N P U T</td>
<td></td>
</tr>
<tr>
<td>bfffb08</td>
<td>53544152</td>
<td>42414449</td>
<td>str1[0-3]</td>
</tr>
<tr>
<td></td>
<td>S T A R</td>
<td>B A D I</td>
<td></td>
</tr>
<tr>
<td>bfffb09</td>
<td>00550408</td>
<td>45053534</td>
<td>str2[4-7]</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>N P U T</td>
<td></td>
</tr>
<tr>
<td>bfffb09</td>
<td>30561540</td>
<td>42414449</td>
<td>str2[0-3]</td>
</tr>
<tr>
<td></td>
<td>0 V...@</td>
<td>B A D I</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10.2 Basic Buffer Overflow Stack Values
Buffer Overflow Attacks

Attacker needs:

- To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker’s control
- To understand how that buffer is stored in memory and determine potential for corruption

Identifying vulnerable programs can be done by:

- inspection of program source
- tracing the execution of programs as they process oversized input
- using tools such as fuzzing to automatically identify potentially vulnerable programs
Stack Buffer Overflows

- Occur when the buffer is located on stack
  - also known as stack smashing
  - exploits included an unchecked buffer overflow

- Widely exploited

- Stack frame
  - when one function calls another it needs somewhere to save the return address
  - also needs locations to save the parameters to be passed in to the called function and to possibly save register values
Shellcode

- **Code supplied by attacker**
  - often saved in buffer being overflowed
  - traditionally transferred control to a user command-line interpreter (shell)

- **Machine code**
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently a number of sites and tools have been developed that automate this process

- **Metasploit Project**
  - provides useful information to people who perform penetration, IDS signature development, and exploit research
Stack Overflow Variants

● **Target program can be:**
  ○ a trusted system utility
  ○ network service daemon
  ○ commonly used library code

● **Shellcode functions**
  ○ launch a remote shell when connected to
  ○ create a reverse shell that connects back to the hacker
  ○ use local exploits that establish a shell
  ○ flush firewall rules that currently block other attacks
  ○ break out of a chroot (restricted execution) environment, giving full access to the system
Buffer Overflow Defenses

- Buffer overflows are widely exploited

- Two broad defense approaches
  - Compile-time: aim to harden programs to resist attacks in new programs
  - Run-time: aim to detect and abort attacks in existing programs
Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - not vulnerable to buffer overflow attacks
  - compiler enforces range checks and permissible operations on variables

- Disadvantages
  - additional code must be executed at run time to impose checks
  - flexibility and safety comes at a cost in resource use
  - distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
  - limits their usefulness in writing code, such as device drivers, that must interact with such resources
Compile-Time Defenses: Stack Protection

- Add function entry and exit code to check stack for signs of corruption
- Use random canary
  - value needs to be unpredictable
  - should be different on different systems
- Stackshield and Return Address Defender (RAD)
  - GCC extensions that include additional function entry and exit code
    - function entry writes a copy of the return address to a safe region of memory
    - function exit code checks the return address in the stack frame against the saved copy
    - if change is found, aborts the program
Run-Time Defenses: Executable Address Space Protection

● Use virtual memory support to make some regions of memory non-executable
  ○ requires support from memory management unit
  ○ long existed on SPARC / Solaris systems
  ○ recent on x86 Linux/Unix/Windows systems

● Issues
  ○ support for executable stack code
  ○ e.g., Java Runtime system
  ○ special provisions are needed
Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - stack, heap, global data
  - using random shift for each process
  - large address range on modern systems means wasting some has negligible impact

- Randomize location of heap buffers

- Random location of standard library functions
Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory
  - flagged in memory management unit as illegal addresses
  - any attempted access aborts process
- Further extension places guard pages between stack frames and heap buffers
  - cost in execution time to support the large number of page mappings necessary
Replacement Stack Frame

- **Variant that overwrites buffer and saved frame pointer address**
  - Saved frame pointer value is changed to refer to a dummy stack frame
  - Current function returns to the replacement dummy frame
  - Control is transferred to the shellcode in the overwritten buffer

- **Off-by-one attacks**
  - Coding error that allows one more byte to be copied than there is space available

- **Defenses**
  - Any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code
  - Use non-executable stacks
  - Randomization of the stack in memory and of system libraries
Return to System Call

- Defenses
  - Any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code
  - Use non-executable stacks
  - Randomization of the stack in memory and of system libraries

- Stack overflow variant replaces return address with standard library function
  - Response to non-executable stack defenses
  - Attacker constructs suitable parameters on stack above return address
  - Function returns and library function executes
  - Attacker may need exact buffer address
  - Can even chain two library calls
Heap Overflow

- **Attack buffer located in heap**
  - Typically located above program code
  - Memory is requested by programs to use in dynamic data structures (such as linked lists of records)

- **No return address**
  - Hence no easy transfer of control
  - May have function pointers can exploit
  - Or manipulate management data structures

- **Defenses**
  - Making the heap non-executable
  - Randomizing the allocation of memory on the heap
Global Data Overflow

- **Defenses**
  - Non executable or random global data region
  - Move function pointers
  - Guard pages

- **Can attack buffer located in global data**
  - May be located above program code
  - If has function pointer and vulnerable buffer
  - Or adjacent process management tables
  - Aim to overwrite function pointer later called
Summary

- **Buffer overflow (buffer overrun)**
  - more input placed into a buffer than the allocated capacity

- **Stack buffer overflows**
  - targeted buffer is located on the stack
  - function call mechanisms
  - stack frame
  - stack overflow vulnerabilities

- **Shellcode**
  - shellcode development
  - position independent
  - cannot contain NULL values

- **compile-time defenses**
  - resist attacks in new programs

- **run-time defenses**
  - detect and abort attacks in existing programs
  - stack protection mechanisms

- **Other forms of overflow attacks**
  - Replacement stack frame
  - Return to system call
  - Heap overflows
  - Global data area overflows
  - Other types of overflows