How to Build an Exploit

CS 166: Introduction to Computer Systems Security
The most common vulnerability?

- Most common type of vulnerability of last 25 years through vulnerability repositories: CVE from MITRE and the NVD from NIST
  
  Yves Younan - RSA 2013

- The winner is ...

Buffer Overflow

for all categories: General, Serious and Critical Vulnerabilities


2/19/16
What is an Exploit?

• An exploit is any input to breach the security of a network or information system in violation of security policy.

• An attack is the intentional act of attempting to bypass one or more security services or controls of an information system.
Buffer Overflow Attack

• One of the most common OS bugs is a buffer overflow
  – The developer fails to check that an input fits into a fixed-size buffer
  – An attacker gives an input which is longer than the buffer
  – The input overwrites a portion of the memory of the process beyond the end of the buffer
  – Causes the application to behave improperly and unexpectedly

• Effect of a buffer overflow
  – The process can operate on malicious data or execute malicious code passed in by the attacker
  – With code execution, to the computer, it looks like that’s what the program wanted to do
Vulnerabilities and Attack Method

• How do we get the program input?
  – The program is owned by root and is setuid – a user on the same machine can execute it and give it input (command-line arguments, stdin, etc)
  – The program is part of a web application, and web users can pass the input as part of their web requests

• Typical attack method
  1. Find vulnerability
  2. Reverse engineer the program
  3. Build the exploit
Buffer Overflow

domain.c
Main(int argc, char *argv[ ])
/* get user_input */
{
    char var1[15];
    char command[20];
    strcpy(command, "whois ");
    strcat(command, argv[1]);
    strcpy(var1, argv[1]);
    printf(var1);
    system(command);
}

- Retrieves domain registration info
- e.g., domain brown.edu

2/19/16
domain.c
Main(int argc, char *argv[])
/*get user_input*/
{
  char var1[15];
  char command[20];
  strcpy(command, "whois ");
  strcat(command, argv[1]);
  strcpy(var1, argv[1]);
  printf(var1);
  system(command);
}

• argv[1] is the user input
• strcpy(dest, src) does not check buffer
• strcat(d, s) concatenates strings
**strcpy() vs. strncpy()**

- Function `strcpy()` copies the string in the second argument into the first argument
  - e.g., `strcpy(dest, src)`
  - If source string > destination string, the overflow characters may occupy the memory space used by other variables
  - The **null character** is appended at the end automatically
- Function `strncpy()` copies the string by specifying the number `n` of characters to copy
  - e.g., `strncpy(dest, src, n); dest[n] = '\0'`
  - If source string is longer than the destination string, the overflow characters are discarded automatically
  - You have to place the **null character** manually
The Unix `fingerd()` system call, which runs as root (it needs to access sensitive files), used to be vulnerable to buffer overflow.

Write malicious code into buffer and overwrite return address to point to the malicious code.

When return address is reached, it will now execute the malicious code with the full rights and privileges of root.
Shellcode Injection

• An exploit takes control of attacked computer so injects code to “spawn a shell” or “shellcode”

• A shellcode is:
  – Code assembled in the CPU’s native instruction set (e.g. x86, x86-64, arm, sparc, risc, etc.)
  – Injected as a part of the buffer that is overflowed.

• We inject the code directly into the buffer that we send for the attack

• A buffer containing shellcode is a “payload”
Buffer Overflow Mitigation

- We know **how** a buffer overflow happens, but **why** does it happen?
- This problem could not occur in Java; it is a C problem
  - In Java, objects allocated dynamically on the heap (except ints, etc.)
  - Also cannot do pointer arithmetic in Java
  - In C, however, you can declare things directly on the stack
- One solution is to make the buffer dynamically allocated
- Another (OS) problem is that **fingerd** had to run as root
  - Just get rid of **fingerd**’s need for root access (solution eventually used)
  - The program needed access to a file that had sensitive information in it
  - A new world-readable file was created with the information required by **fingerd**
Stack-based buffer overflow detection using a random canary

Normal (safe) stack configuration:

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Other local variables</th>
<th>Canary (random)</th>
<th>Return address</th>
<th>Other data</th>
</tr>
</thead>
</table>

Buffer overflow attack attempt:

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Overflow data</th>
<th>Corrupt return address</th>
<th>Attack code</th>
</tr>
</thead>
</table>

• The canary is placed in the stack prior to the return address, so that any attempt to over-write the return address also over-writes the canary.
No eXecute (NX) bit

- Modern processors support a feature called no eXecute bit
  - Allows a system to control the execution of given segments of memory
  - Stack, Heap, Data are flagged as non-executable
  - Text is flagged as non-writeable
- NX helps prevent certain types of buffer overflow exploits from working as expected.
**ASCII Armour**

**Address Space**

- The idea is to load libraries in the 16 first megabytes of the address space.
- Load code into addresses containing NULL byte (\'0\') in the range 0x00 (0x00****** - 0x0100******)
- Makes it hard to construct address or pass arguments by exploiting string functions (i.e. Strcpy)
- Fedora uses this protection
ASLR

- Address Space Layout Randomization
- Involves randomly arranging the positions of key memory segments:
  - base of the executable, position of libraries, base address of stack, heap, text, etc.
- Randomization can be done at different time, or by rewriting existing binaries

**Attack:** guess pseudo-random number generator by repeated execution