Buffer Overflow Attacks

Sources:

- *Smashing the Stack for Fun and Profit* by Aleph One, Phrack, Vol. 7, No. 49
- *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade* by C. Cowan, SANS 2000
- *CERT Vulnerability Note VU#907819*, www.kb.cert.org
- *Hackers Beware* by E. Cole
- *Windows 2000 System Monitor ActiveX control buffer overflow*, www.iss.net
- *Statically Detecting Likely Buffer Overflow Vulnerabilities* by D. Larochelle and D. Evans, USENIX 2001
- *Building Secure Software* by J. Viega and G. McGraw

- *2011 CWE/SANS Top 25 Most Dangerous Software Errors*, cwe.mitre.org/top25/#CWE-120

Introduction

A *buffer overflow* (*BOF*) occurs when:

- A buffer allocated to store input to a program is not large enough to store data fed to the program.
- Consequently, the program writes past the end of the buffer.

The goal of a BOF attack is usually to give the attacker *control of a privileged program*.

Typically, a **`root`** program is attacked to execute code like **`exec(sh)`** to get a root shell.

Subgoals:

1. Arrange for attack code to be available in program's address space
2. Get the program to jump to that code, with parameters loaded into registers and memory

Buffer overflows accounted for more than 50% of security bugs resulting in CERT advisories in 1999.

Buffer overflows usually affect programs written in languages *without automatic array bounds checking*, such as C and C++.

Some standard C functions are susceptible to buffer overflows because they don't check array bounds.

Examples [Viega & McGraw]:

- `strcpy()`
- `strcat()`
- `sprintf()`
- `scanf()`
- `sscanf()`
- `fscanf()`
- `vfscanf()`
- `vsprintf()`
- `vscanf()`
- `vsscanf()`
- `streadd()`
- `strecpy()`
- `strtrns()`

With these functions, it is up to the programmer to do bounds checking.

Example [Viega & McGraw]:

```
if (strlen(src) >= dst_size) {
    /* Do something appropriate, like throw an
    error. */
}
else {
    strcpy(dst, src);
}
```

However, programmers often fail to do such checking.

In general, it is prudent to assume that other people's code does *not* do proper bounds checking.

Placing Attack Code in Program's Address Space

There are two ways to do this:

Inject the code:

- Attacker provides input string which program stores in buffer
- String contains bytes that are native CPU instructions

Use code already there:

- Often, the required code is already present in the program's address space, e.g.,
  - Attack code needs to execute "**`exec("/bin/sh")`**"
  - There is code in **`libc`** that executes "**`exec(arg)`**"
  - Attacker can make pointer point to "**`/bin/sh`**" and jump to appropriate instructions in **`libc`**

Causing the Program to Jump to the Attacker's Code

Basic method is to *overflow* a buffer so as to corrupt an adjacent part of the program's state.

The adjacent state can be overwritten with almost any sequence of bytes.

This can bypass the programming language type system and the victim program's logic.

Most BOFs seek to corrupt *pointers to code*, e.g.,

- *Return address* of function or procedure, stored in *activation record* on call stack, is overwritten with address of attack code

- *Function pointers*: attacker needs to find overflowable buffer adjacent to function pointer in stack, heap, or static data area

  - Later call through function pointer will cause jump to attack code

Memory Regions

Regions of memory used by programs [Viega & McGraw]:

- *Program arguments* and the *program environment*.
- The *call stack*. It typically grows down, toward the heap.
- The *heap*. It typically grows up, towards stack.
- The *data segment* contains initialized globally available data.
- The *block storage segment* (*BSS*) contains globally available data not initialized until **`main()`** is called.
- The *text segment* contains read-only program code.

Buffer overflows can occur in any of these regions.

Heap storage is allocated using functions like *malloc* and *new*.

Stack allocation occurs automatically whenever a function is called.

An *activation record* or *stack frame* is pushed onto the stack to hold information about the current call.

The contents of a stack frame are dependent on the machine architecture and compiler.

Common items:

- Values of (nonstatic) local variables
- Actual parameters
- Saved register information
- *Return address*

Stack Overflows

Heap and static buffer overflows are generally more difficult to exploit than stack overflows because of the difficulty of finding and overwriting security-critical variables.

Outline of stack overflow attack [Viega & McGraw]:

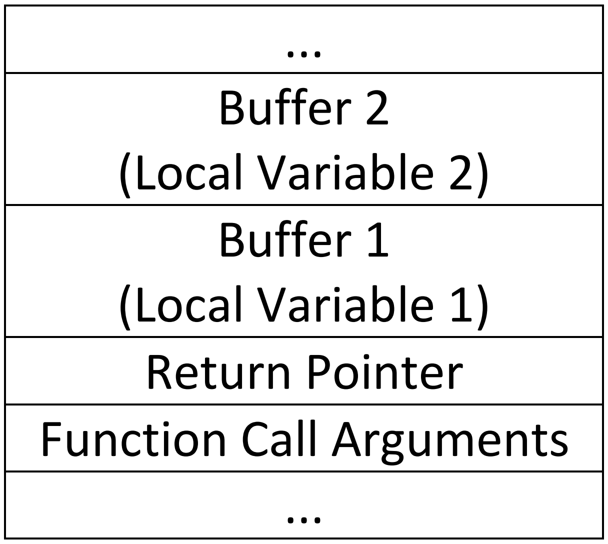
1. Find a stack-allocated buffer to overflow.

2. Place some exploit code in (stack) memory.

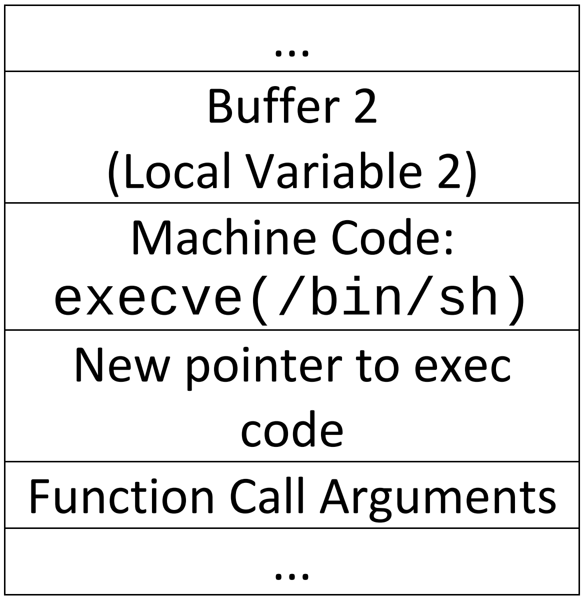
3. Write over the return address on the stack with a value that causes the program to jump to the exploit code.

Program is usually fed large string that changes activation record *and* contains attack code.

Normal stack [Cole]:



“Smashed” Stack:



It is necessary to “map” the stack for a particular function on a particular platform.

This can be done with the aid of a test program.

Example [Viega & McGraw]:

```
char *j;
int main();

void test(int i) {
  char buf[12];
  printf(“&main = %p\n”, &main);
  printf(“&i = %p\n”, &i);
  printf(“&buf[0] = %p\n”, buf);
  for (j = buf – 8; j < ((char *)&i) + 8; j++)
    printf(“%p: 0x%x\n”, j, *(unsigned char *) j);
}

int main() {
  test(12);
}
```

Example: Stack frame format (x86, Linux?, GCC?) [Viega & McGraw]:

Low address

Local variables

Old base pointer

Return address

Parameters to function

High address

Privilege Elevation

If a program being exploited runs with a high privilege level, the attacker *inherits* that privilege.

Examples of UNIX programs that use *setuid* to perform tasks as root level [www.acm.uiuc.edu]:

- **login** - Needs it to set itself up for a user.
- **passwd** - Allows a user to change their password. This may not be necessary.
- **rcp, rlogin, etc** - If you want people to use the r-commands then the suid bit must remain. The r-commands need to bind to a privileged port which requires uid 0.
- **su** - If you want people to su, then leave it on. At the very least restrict it to a certain group.
- **ufsdump** - Unless you want a non-root user doing backups get rid of the suid bit.

Shell-launching code can be obtained from the Web.

Example: Linux on Intel Machines [Aleph]:

```
void main() {
__asm__("
     jmp    0x1f                    # 2 bytes
     popl   %esi                    # 1 byte
     movl   %esi,0x8(%esi)          # 3 bytes
     xorl   %eax,%eax               # 2 bytes
     movb   %eax,0x7(%esi)          # 3 bytes
     movl   %eax,0xc(%esi)          # 3 bytes
     movb   $0xb,%al                # 2 bytes
     movl   %esi,%ebx               # 2 bytes
     leal   0x8(%esi),%ecx           # 3 bytes
     leal   0xc(%esi),%edx          # 3 bytes
     int    $0x80                    # 2 bytes
     xorl   %ebx,%ebx               # 2 bytes
     movl   %ebx,%eax                # 2 bytes
     inc    %eax                    # 1 bytes
     int    $0x80                   # 2 bytes
     call   -0x24                   # 5 bytes
     .string \"/bin/sh\"            # 8 bytes
                                    # 46 bytes total
");
```

Example: AOL Instant Messanger Buffer Overflow [CERT, Cole]

A remotely exploitable buffer overflow existed in the AOL Instant Messaging Client for Windows.

It occurred when parsing messages from another user inviting the victim to participate in a game.

More specifically, it occurred in parsing a Type, Length, Value (TLV) tuple with type 0x2711.

The following versions are vulnerable:

- AIM for Windows, version 1.0 - 3.0.1415
- AIM for Windows, version 4.3.2229 and greater

The buffer overflow allowed a remote attacker to execute arbitrary code on the victim's system.

Windows 2000 System Monitor ActiveX Control Buffer Overflow [Cole, iss.net]

The System Monitor ActiveX control in Microsoft Windows 2000 was vulnerable to a buffer overflow in the "LogFileName" field.

By using special parameters to invoke this control, a malicious Web site operator could execute arbitrary code on a visiting user's computer.

Countermeasure: patch.

Protecting Against Buffer Overflow Attacks

Actions [Cole]:

- Close port or service.
- Apply vendor’s patch or install latest version of software.
- Filter specific traffic at firewall.
- Test key applications.
- Run software at least privilege required.

Static code analysis may used to detect buffer overflow vulnerabilities [Larochelle and Evans].

CWE BOF Preventions & Mitigations

- Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
  - Languages that perform their own *memory management*, such as Java and Perl, are not subject to buffer overflows

- Use a *vetted library or framework* that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
  - Examples include the Safe C String Library (SafeStr) by Messier and Viega, and the Strsafe.h library from Microsoft.

- Run or compile your software using features or extensions that automatically provide a *protection mechanism* that mitigates or eliminates buffer overflows.
  - Certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code: the Microsoft Visual Studio /GS flag,

    Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice.

- For any security checks that are performed on the client side, ensure that these checks are *duplicated on the server side*, in order to avoid CWE-602.
    - Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely.

- Make data segment of victim program's address space *non-executable.*

- Use a feature like *Address Space Layout Randomization* (*ASLR*).

Examples: StackGuard and PointGuard

StackGuard is a compiler technique for providing *code pointer integrity checking* to the return address of function activation records.

It is implemented as an enhancement to the GCC code generator for emitting code to *set up and tear down function calls*.

The enhanced setup code places a *canary word* next to the return address on the stack.

The enhanced tear-down code *checks if the canary* word is intact before jumping to the return address.

Random canaries are used so as to prevent forgeries.

PointGuard generalizes StackGuard to place canaries next to *all code pointers*.