Offensive technologies
Fall 2017

Lecture 5 – Creating Shell Code

Fabio Massacci

Ethical Acceptance

- You are bound by the terms and conditions of this course
  - You try offensive technologies **only** in the lab
  - You are **not allowed** to disclose information about any individual that you find during the analysis
  - Your final deliverable, as approved by the professor is **the only public deliverable** you are allowed to disclose to third parties
- Any use outside the agreed framework of the course may be penally relevant (i.e. a crime)
  - Everything is **isolated** from rest of infrastructure → you must deliberately exfiltrate material → cannot claim that "happened by mistake"
  - The same considerations apply if you give material to other students who have not signed the agreement → aiding and abetting = same penal responsibility as if you did it yourself.

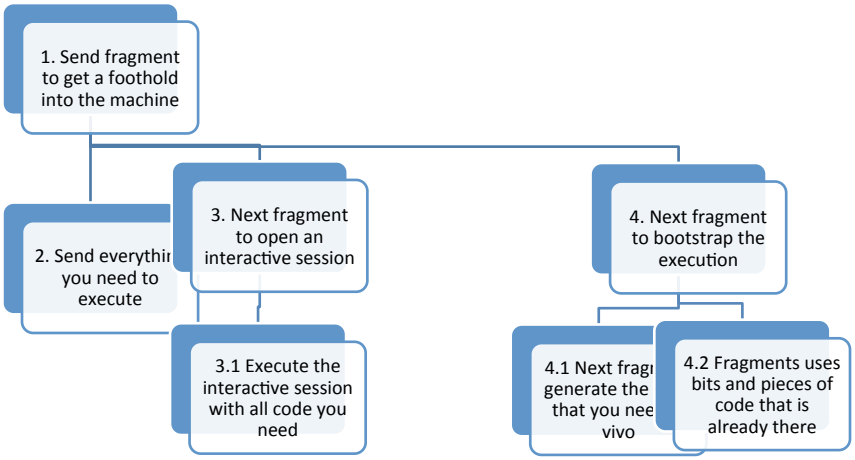
Key Idea of any type of exploit

- Send data to a complex web/stack of applications
- Some bug in the data processing flow makes a component take an unexpected turn
- If you are lucky the wrong turn of control is all you need
  - Example: access control error → grant you access with admin privileges on a web server
- Else the data itself must be executed to transfer you the control
  - If you are lucky the data is directly executable at the wrong turn
    - Example: SQL injection (or JS Eval) → data is directly executed by the DB (or WS)
  - Else you must somewhat encode it in the data
    - Example: Buffer Overflow and shell code injection

Turing Completeness

- Theoretical constructs needed to execute an arbitrary function?
  - Array of memory locations storing integers
  - Test on zero on memory locations
  - Addition of values in memory locations
    - Subtraction makes life easier but it is not necessary
  - While loop
    - Typically implemented by conditional jumps
  - Assignments to and from memory locations
- Additional practical requirements
  - Assignments to and from registers
    - Modern architectures don't usually make math operations on memory locations
  - Locating addresses of functions in memory
    - Otherwise you can't jump on them
  - Deploying functions in memory
  - Initializing EIP (Instruction Pointer)
    - The computer needs to know where to start

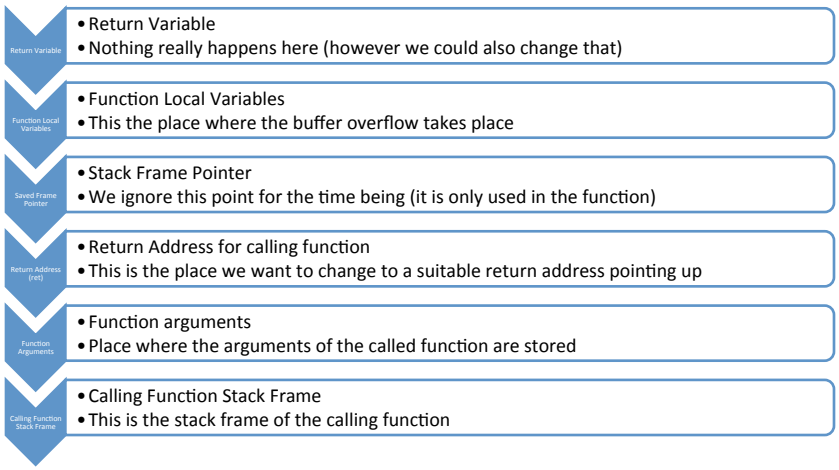
How to ship the code to the machine



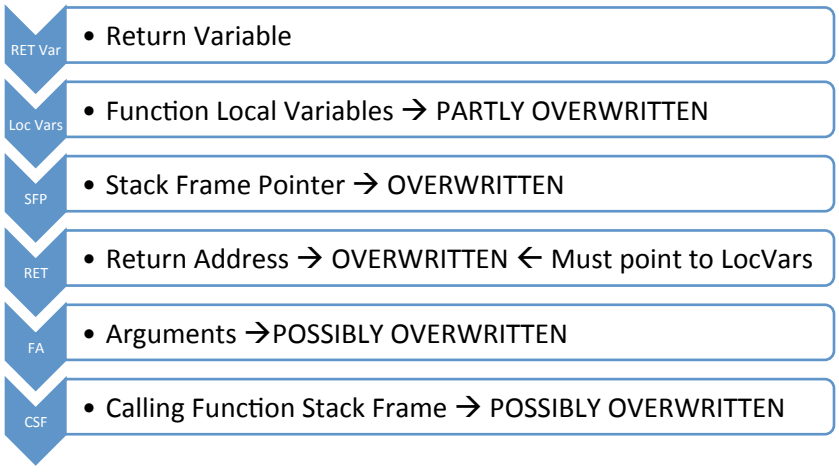
Registers

- eax, ecx, edx, ebx → general purpose registers
  - Accumulator, Counter, Data, Base registers
  - Basically for everything (not really true, some operands used them in especial ways)
- esp, ebp → stack and base pointers
- esi, edi → source index, destination index
  - Used to copy things in bulk, some instruction needs them but for us they are just general registers
- eip → instruction pointer
  - Where the next instruction is
- eflags → several bit flags used for comparison and memory segmentation
  - This is only for very advanced users and I would need to look out for what they are

Stack Grows Downward (i.e. higher addresses are at the bottom)



What happens with Overflow



Making shell code more robust

- Difficulties
  - Shellcode may be either too small or too large
  - Return address might not be precisely known
- We extend the shell code
  - «Bottom» of the code with NOP (\x90)
  - Shell code we want to execute
  - «Top» of the code with copies of the tentative return address
  - SHELLCODE=$(perl -e 'print "\x90" x 200')$(cat shellcode.txt)$(perl -e 'print "return address in exa" x 40‘)
    - Beware return address for little endian → address 0x08048d70 → string \x70\x8d\x04\x08
    - Ret on «top» of the code will actually point downward so eip will move upward
- How do we know the return address?
  - Well, try and experiment (ASLR makes things difficult) → Chap. 0x330 in book

Other places to overflow

- We don’t always need to send shellcode through a buffer in the stack
- Environmental Variables are helpful
  - We need ret address to point at the environmental variable where the shellcode is
  - Always at the beginning of the stack → very little uncertainty on where they are!
- System calls that require enviroments are also helpful
  - execle(comm,args[],env[])
  - env[] is list of environmental variables terminated by a NULL pointer
  - Now if shellcode is an environmental variable and we make sure it is terminated by a null pointer → every other var will be ignored → shellcode will be last var
    - We only need to put the return address in the buffer and this deterministically starts from the bottom of the stack of the process

More places to overflow

- Heap
  - There are places to return control but progressively harder
  - Simple mechanism is to actually change files/environmental variables/ etc that are manipulated by the victim
    - Write to /etc/passwd instead of /tmp/printer-spooler
    - Change argument to a legitimate system call so that argument is $(execute this)
- bbs
  - Contains global variables, for example function pointers
  - If a global variable could be overflown → function pointer below it can be overflown
    - Either do wrong thing with an existing function → cannot be prevented in any way by the OS protection measures eg unexecutable stack (why?)
    - Or Executing shellcode from either input or shell variable
- .dtors (destructuctors)
  - Functions called to cleanup for program functions after main exit → normally writable
- Global Offset Table → shared libraries
  - Program linking table is write only but it jumps to pointers to address
  - Addresses are in the GOT and they can potentially be rewitten (eg the exit() function

Self-locating code

- We often need the absolute address in the program
  - For example to jump at an address
  - Equally absolute pointers are needed for accessing constant strings, data etc.
- BUT Shellcode is injected into the program
  - it cannot be «linked», memory layout has not be calculated by the compiler/ loader, we don't know absolute addresses in general etc.
  - Need to be position independent
- We use the stack's operations kindly provided by the CPU
  - call <location> → call a function and jump to the address in the <location> operand (absolute or relative), the address of the instruction after the call is pushed to the stack
    - Actual interpretation → «whatever X is in the memory address after a call is interpreted as an instruction whose ABSOLUTE address is pushed into the stack as a return address»
      - This make sense as we are not supposed to have a call to a function in the middle of data, aren't we?
    - We don't need to know the address now, the OS will do it for us
  - Ret → pop the return address from the stack → we got an absolute address to «whatever X»

Example of linkable vs self-locating programs

Linkable Hello World

```
BITS 32                 ;tell assembler is 32 BITS
section .data           ;Data segment
msg db "Hello, world!", 0x0a, 0xd ; string, newline, carriage return
section .text           ;Code segment
global _start           ;default entry point for ELF linking

_start:
        mov ecx, msg            ;put address of string in ecx
        mov eax 4               ;write is syscall #4
        mov ebx, 1              ;stdout is 1
        mov edx, 15             ;put length of string in edx
        int 0x80                ;SYSCALL: write(1,msg,14)
mov eax, 1              ;exit is syscall #1
mov ebx, 0              ; exit with success
int 0x80                ;SYSCALL exit(0)
```

Self-locating Hello World

```
BITS 32                 ;tell assembler is 32 BITS
                        ;no explicit data segment
                        ;no explicit code segment
                        ;no entry point for ELF linking
call myself
db "Hello, world!", 0x0a, 0xd           ; string, newline, carriage return
myself:
        pop ecx         ;load address of string in ecx
        mov eax 4       ;write is syscall #4
        mov ebx, 1      ;stdout is 1
        mov edx, 15             ;put length of string in edx
        int 0x80        ;SYSCALL: write(1,msg,14)
mov eax, 1              ;exit is syscall #1
mov ebx, 0              ; exit with success
int 0x80                ;SYSCALL exit(0)
```


Polyglot's misinterpretation goes both ways

- General Problem with Polyglots
  - Hacker compile «code» into «data» → send «data» to Victim(s) → Victim 1 pass «data» to Victim 2 → … → Victim n (mis)interpret it as «code» and **executes** it
  - Hacker compile «code» into «data» → send «data» to Victim(s) → Victim 1 pass «data» to Victim 2 → … → Victim j (mis)interpret it as «data» and **corrupts** it… → Victim n interpret it as «code» but code has been corrupted…
- When shipping shellcode around → it is interpreted as strings → null byte stops the string → any 0ed byte is interpreted as a null string and the shell code is abruptly truncated
  - Example: shell code is small → jumps are small but address are on 32+ bits → leading zeros create null bytes when reverted in little endian → ahi ahi
  - Another example "mov eax, 0x4" = "B8 04 00 00 00" in binary → as soon as the input analyzer arrives to the 00s the code is truncated
- Similar issues with other executable languages (eg escape characters)

Various solutions to avoid null bytes

- Make short jumps
  - So the program won't fill with leading zeros
  - Trade-off can jump at most 128bytes either way → ok shell code is short
- Use two-complement
  - jump at the end of the program (higher address)
  - jump backward so this is a small negative number → leading bit turned on → 0xff (little endian)

This shellcode should work… but doesn't

Linkable Hello World

```
call myself                ; many zeros
db "Hello, world!", 0x0a, 0xd
myself:
    pop ecx          ;load address of string in ecx
    mov eax 4        ;write is syscall #4
    mov ebx, 1             ;stdout is 1
    mov edx, 15      ;put length of string in edx
    int 0x80         ;SYSCALL: write(1,msg, 14)
mov eax, 1                 ;exit is syscall #1
mov ebx, 0                 ; exit with success
int 0x80                   ;SYSCALL exit(0)

;
```

Self-locating Hello World

```
jmp short endofprogram

myself:
    pop ecx          ;load address of string in ecx
    mov eax 4        ;write is syscall #4
    mov ebx, 1       ;stdout is 1
    mov edx, 15            ;put length of string in edx
    int 0x80         ;SYSCALL: write(1,msg,14)
mov eax, 1           ;exit is syscall #1
mov ebx, 0           ; exit with success
int 0x80             ;SYSCALL exit(0)
endofprogram:
    call myself
    db "Hello, world!", 0x0a, 0xd
```

More zeros and more ways to avoid null bytes

- Use direct operations for low/high bytes of a register
  - Registers are 32-bits and originally were 16bits so lots of trailing is put to zero.
    - mov eax, 0x4 → B8 04 00 00 00
    - mov ax, 0x4 → 66 B8 04 00
    - mov al, 0x4 → B0 04 → ok, but the remaining three bytes are arbitrary → must be zeroed
- Zero a register by using aritmetics or logic
  - mov eax, 0x11223344 → B8 44 33 22 11
  - sub eax, 0x11223344 → 2D 44 33 22 11
  - sub eax, eax → 29 C0
  - xor eax, eax → 31 C0 → this is considered the best (but…)

Removing zeros

```
jmp short endofprogram

myself:
        pop ecx         ;load address of string in ecx
        mov eax 4               ;write is syscall #4
                        BUT 4 = 0x4000
        mov ebx, 1              ;stdout is 1
                        BUT 1 = 0x1000
        mov edx, 15             ;put length of string in edx
                        BUT 15 =…
        int 0x80                ;SYSCALL: write(1,msg,14)
mov eax, 1                      ;exit is syscall #1
mov ebx, 0                      ; exit with success
int 0x80                        ;SYSCALL exit(0)
endofprogram:
        call myself
        db "Hello, world!", 0xa, 0xd
```

```
jmp short endofprogram

myself:
        pop ecx         ;load address of string in ecx
        xor eax, eax            ;first zero the register
        mov al, 4               ;Now 4 is byte = 0x4
        xor ebx, ebx            ;stdout is 1
        inc ebx
        xor edx,edx             ;put length of string in edx
        mov dl, 15
        int 0x80                ;SYSCALL: write(1,msg,14)
mov al,1                        ;exit syscall #1, top 3 bytes = 0x0,
why?
dec ebx                 ; decrement ebx back down
int 0x80                        ;SYSCALL exit(0)
endofprogram:
        call myself
        db "Hello, world!", 0xa, 0xd
```

Executing everything

- The process can be repeated for different syscalls
  - Find syscall number
    - #11 for execve(filename,argv,envp)
    - #164 (0xa4) for setresuid(0,0,0) which restore all root privileges
    - #102 (0x66) for sockcall
  - Load it
    - Syscall number in eax
    - Value (or pointer) to first argument in ebx, second in ecx, third in edx etc.
  - Call int 0x80
- What if we need to null terminate something (eg a string)?
  - xor eax, eax
  - mov [ebx+n], eax
    - use ax or al depending on how many bits we need to zero
- What if we need to load an address to something that is in the middle of something else
  - lea ecx, [ebx+n] → loads the effective address of what is at the address ebx+n
- Can also use push to add things to the stack instead of jumping

Blocking polyglots

- A firewall/sanitizer can filter anything that is not correct data
  - Hacker compile «code» into «data» → send «data» to Victim(s) → Victim 1 pass «data» to Victim 2 → … → Victim n (mis)interpret it as «code» and **executes** it
  - Hacker compile «code» into «data» → send «data» to Victim(s) → Victim 1 pass «data» to Victim 2 → … → Firewall j drops anything that is NOT looking as «data» → Victim n no longer gets «code»…

How to bypass «language checks»

- General (automated) solution is cross compilation
  - Identify a «conformant instruction set» that i) satisfies the «data» language check and ii) corresponds to a Turing complete set
    and eax, 0x454e4f4a
    and eax, 0x3a313035 ←→ xor eax, eax ←→ mov eax, 0x0
  - Compile any target instruction into the appropriate sequence of «conformant instructions»
    - ALWAYS works but you need to have attended a course on Compilers to do it properly
    - May increase the lenght of the shellcode in general (as you are not optimizing the compiler)
- On-line Construction
  - Load a «conformant loader»
  - Allocate some space in memory
    - eg for stack extend the ESP to make space for the shellcode

Additional Reading

- Jon Erickson, «Hacking, The art of exploitation», No Starch Press 2° edition
  - The source used in the book can be downloaded free of charge from the publisher's web site
- For the general idea of a compiler for «printable shellcode»
  - See Chapter 3 of Pieter Philippaerts "Security of Software on Mobile Devices", PhD Thesis
  - https://lirias.kuleuven.be/bitstream/123456789/274862/1/Thesis_Final-with_cover.pdf

Useful commands for exercises in the book

- gcc -g -o program program.c
- objdump -M intel -D program | grep –A20 function\>:
  - spits out the next 20 lines after <function>: in the assembly
- gdb –q
  - set disassembly intel
  - break *function*
  - run
  - info register(s) *register*
  - disassemble *function*
  - *x/x,u,t,i,s,c location $location*
    - *X exadecimal, t binary, i instruction*
    - *n to make sequences (b,w to change size)*
  - *nexti, continue*
  - *$variable, &variable*
- *$(perl -e 'print "A" x 20 . "\x14" x 5 . "ABCD";')*
- for line in $(cat shellcode.txt); do echo –en $line; done > shellcode.bin
- export SHELLCODE=$(perl -e 'print "\x90" x 200')$(cat shellcode.txt)$(perl -e 'print "return address in exe" x 40')
- echo "set dis intel" >> ~/.gdbint
- echo 0 > /proc/sys/kernel/randomize_va_space
- hexdump –C shellcode.bin
- nasm –f elf helloworld.asm
- ld –o helloworld helloworld.o
- ndisasm –b32 helloworld
- less /usr/include/ams-i386/unistd.h
- sudo apt install nasm
- sudo mount -t vboxsf -o uid=$UID,gid=$(id -g) globalshare ~/mycopy