Offensive Software Exploitation

SEC-300-01/CSI-301-02

Ali Hadi

@binaryz0ne

Jumping Strategies…

Jumping around in memory space…

Jumping Strategies

- Using “jmp esp” was an almost perfect scenario
- Not that ‘easy’ every time!
- Let’s check what other ways to execute/jump to shellcode
- Also, what if you are faced with small buffer sizes!

JMP (or CALL)

Cited [1]

jump (or call) a register that points to the shellcode.

- With this technique, you basically use a register that contains the address where the shellcode resides and put that address in EIP.
- You try to find the opcode of a “jump” or “call” to that register in one of the dll’s that is loaded when the application runs.
- When crafting your payload, instead of overwriting EIP with an address in memory, you need to overwrite EIP with the address of the “jump to the register”.
- Of course, this only works if one of the available registers contains an address that points to the shellcode.

POP RET

Cited [1]

pop return

- If none of the registers point directly to the shellcode, but you can see an address on the stack (first, second, … address on the stack) that points to the shellcode
- You can load that value into EIP by first putting a pointer to pop ret, or pop pop ret, or pop pop pop ret (all depending on the location of where the address is found on the stack) into EIP.

PUSH RET

Cited [1]

push return

- this method is only slightly different than the “call register” technique
- If you cannot find a <jump register> or <call register> opcode anywhere, you could simply put the address on the stack and then do a ret
- So you basically try to find a push <register>, followed by a ret
- Find the opcode for this sequence, find an address that performs this sequence, and overwrite EIP with this address

JMP [reg + offset]

Cited [1]

jmp [reg + offset]

- If there is a register that points to the buffer containing the shellcode, but it does not point at the beginning of the shellcode, you can also try to find an instruction in one of the OS or application dll's, which will add the required bytes to the register and then jumps to the register

Blind Return

Cited [1]

blind return

- We know that ESP points to the current stack position (by definition)
- A RET instruction will ‘pop’ the last value (4bytes) from the stack and will put that address in ESP
- So if you overwrite EIP with the address that will perform a RET instruction, you will load the value stored at ESP into EIP
- If you are faced with the fact that the available space in the buffer (after the EIP overwrite) is limited, but you have plenty of space before overwriting EIP, then you could use jump code in the smaller buffer to jump to the main shellcode in the first part of the buffer

SEH

Cited [1]

Structured Exception Handler (SEH)

- Every application has a default exception handler which is provided for by the OS.
- So even if the application itself does not use exception handling, you can try to overwrite the SEH handler with your own address and make it jump to your shellcode
- Using SEH can make an exploit more reliable on various windows platforms, but it requires some more explanation before you can start abusing the SEH to write exploits
- The idea behind this is that if you build an exploit that does not work on a given OS, then the payload might just crash the application (and trigger an exception)

SEH – Cont.

Cited [1]

- So if you can combine a “regular” exploit with a seh based exploit, then you have build a more reliable exploit
- Just remember that a typical stack based overflow, where you overwrite EIP, could potentially be subject to a SEH based exploit technique as well, giving you
  - more stability
  - a larger buffer size
  - overwriting EIP would trigger SEH

POPAD

Cited [1]

popad (pop all double) will pop double words from the stack (ESP) into the general-purpose registers, in one action (opcode = 0x61)

- Loaded order: EDI, ESI, EBP, EBX, EDX, ECX and EAX
- As a result, the ESP register is incremented after each register is loaded (triggered by the popad)
- One popad will thus take 32 bytes from ESP and pops them in the registers in an orderly fashion
- So suppose you need to jump 40 bytes, and you only have a couple of bytes to make the jump, you can issue 2 popad's to point ESP to the shellcode
  - Don't forget to place NOPs at the beginning

Short Jumps

Cited [1]

- In the event you need to jump over just a few bytes, then you can use a couple ‘short jump’ technique to accomplish this
- Short jump (jmp) opcode is 0xeb
- All you need to do is jmp followed by the number of bytes
- So if you want to jump 30 bytes, the opcode is 0xeb,0x1e

Conditional Jumps

Cited [1]

- Conditional (short/near) jump: ("*jump if condition is met*")
- This technique is based on the states of one or more of the status flags in the EFLAGS register (CF,OF,PF,SF and ZF)
- If the flags are in the specified state (condition), then a jump can be made to the target instruction specified by the destination operand
- This target instruction is specified with a relative offset (relative to the current value of EIP)

Conditional Jumps – Cont.

Cited [1]

Example:

- Suppose you want to jump 6 bytes
- Have a look at the flags, and depending on the flag status, you can use one of the opcodes below
- Let's say the Zero flag is 1, then you can use opcode 0x74, followed by the number of bytes you want to jump (0x06 for this case)

Backward Jumps

Cited [1]

- In the event you need to perform backward jumps
  - jump with a negative offset
- Get the negative number and convert it to hex
- Take the DWORD hex value and use that as argument to a jump
  - \xEB or \xE9

Backward Jumps: Example #1

You want to jump backwards 7 bytes

- Assembly instruction is: JMP -7
- OPCode/bytecode (5 bytes): E9F4FFFFFF
- Result would be: "\xE9\xF4\xFF\xFF\xFF“
  - When using JMP this way, it is actually performing a long distance jump and that is why it is using 5 bytes not two.
  - You can notice that from the \xE9 bytecode used at the beginning.

Backward Jumps: Example #2

But what if you only want to jump to a near location with less bytes? Then you can use the following syntax:

- Assembly instruction is: JMP short -7
- OPCode/bytecode (2 bytes): EBF7
- Results would be: "\xEB\xF7“

  – When using JMP this way, you can only jump backwards and forwards in a limited number of bytes. That is why sometimes even if you type JMP -400 in NASM, it will be converted to a long or far distance jump as if you typed JMP LONG -400.

Backward Jumps: Example #3

I guess you know this by now! Jumping backwards 400 bytes:

- You cannot do this with a short jump, so you will either type:

  JMP -400

  Or

  JMP LONG -400

- Result = "\xE9\x6B\xFE\xFF\xFF"

  - As you can see, this opcode is 5 bytes long!
  - Sometimes, if you need to stay within a DWORD size (4 byte limit), then you may need to perform multiple shorter jumps in order to get where you want to be…!

Weird Relative Backward Jump ☺

Cited [1]

"\x59\xFE\xCD\xFE\xCD\xFE\xCD\xFF\xE1\xE8\xF2\xFF\xFF\xFF“

- Explanation

\x59	*POP ECX*
\xFE\xCD	*DEC CH*
\xFE\xCD	*DEC CH*
\xFE\xCD	*DEC CH*
\xFF\xE1	*JMP ECX*
\xE8\xF2\xFF\xFF\xFF	*CALL [relative -0D]*

- Could be adjusted to fit your needs

Summary

- Explained different jumping strategies

References

- Peter “Corelanc0d3r”, Exploit Writing (Jumping to Shellcode), http://www.corelan.be/,
- Memory Corruption 101, NYU Poly, Dino Dai Zovi
- Grayhat Hacking: The Ethical Hacker’s Handbook, 3rd Edition
- The Shellcoders Handbook