English
Shellcode
English Shellcode
**What is Shellcode?**

Shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine. Shellcode is commonly written in machine code, but any piece of code that performs a similar task can be called shellcode.

**Example**

```
SEGMENT .text
    mov eax, eax
    mov al, 29
    int 80h

; put the value 29 into eax (syscall number of pause)
; enter kernel mode

Disassembly of section .text:
00468100 <.text>
00468100: 31 c0          mov eax, eax
00468102: 09 1d          mov $1d, eax
00468104: 0f 80          int $0x80

const char pause_shell[]="\x61\x6f\x6f\x6d\x6f\x6f\x6e\x65\x73\x75\x6c\x6c\x65\x73\x74\x69\x6e\x67\x6f\x6f\x66\x65\x73":
main() {
    (*shell)();
    int (*shell)();
    shell=pause_shell;
    shell();
}
```
What is Shellcode?

A shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine. Shellcode is commonly written in machine code, but any piece of code that performs a similar task can be called shellcode.
Example

SEGMENT .text
  xor eax, eax ; zero eax
  mov al, 29 ; put the value 29 into eax (syscall number of pause)
  int 80h ; enter kernel mode

Disassembly of section .text:
08048100 <.text>:
  8048100:   31 c0     xor %eax,%eax
  8048102:   b0 1d     mov $0x1d,%al
  8048104:   cd 80     int $0x80

const char pause_shell[]="\x31\xc0\xb0\x1d\xcd\x80";

main()
{
  int (* shell)();
  shell=pause_shell;
  shell();
}


Objective

• To demonstrate a technique for automatically producing English Shellcode, transforming arbitrary shellcode in assembly to a representation superficially similar to English prose.
• Does not require an external loader and executes as valid x86 code
• Generates encodings complete with decoder that remains faithful to the corpus and at the same time is statistically similar to the original shellcode

what does an attacker want?

• Code injection
• Gain control of a machine’s program counter
• Common component. Payload/injected code
• Shellcode can be delivered prior to triggering the exploit
Objective

- To demonstrate a technique for automatically producing English Shellcode, transforming arbitrary shellcode in assembly to a representation superficially similar to English prose.
- Does not require an external loader and executes as valid IA32 code
- Generate encodings complete with decoder that remains faithful to the corpus and at the same time is statistically similar to the original shellcode
what does an attacker want?

- Code injection
- Gain control of a machine's program counter
- Common component: Payload/injected code
- Shellcode can be delivered prior to triggering the exploit
The Arms Race

Encoding engines such as the one by Metasploit
Converting any arbitrary payload to alphanumerics
characters

what does this mean to the attacker?

- Alphanumeric shellcode can be stored as directory
  names or paths
- Smaller set of characters than UNICODE
- Has to produce encoding in alphanumeric format and
  use a decoder while maintaining arbitrary encoding

what measures have been taken?

- Single or multi-byte shellcode might be detected
- Antivirus detection of user input, system memory or network
  traffic that seems executable

But the attacker is clever!!!

Use polymorphism
what measures have been taken?

- Directly executable shellcode (machine code) can be detected
- Automated inspection of user input, system memory or network traffic that seems executable

But the attacker is clever!!!

Use polymorphism
The Arms Race

Encoding engines such as the one by Metasploit
Converts any arbitrary payload to alphanumeric characters

What does this mean to the attacker?

- Alphanumeric shellcode can be stored as directory names or paths
- Smaller set of characters than UNICODE
- Has to produce encoding in alphanumeric format and use a decoder while maintaining arbitrary encoding
<table>
<thead>
<tr>
<th>ENCODING</th>
<th>HEX</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>31DB5343536A ...</td>
<td>1#SCSj#jfx########CRfh\fS##jfxPQV####...</td>
</tr>
<tr>
<td>PexAlphaNum</td>
<td>515A56545836 ...</td>
<td>QZVTX630VX4A0B6HH0B30BCVX2BDBH4A2AD...</td>
</tr>
<tr>
<td>Alpha2</td>
<td>374949515A6A ...</td>
<td>7IIQ2jJX0B1FABkBAZB2BA2AA0AAX8BBPux...</td>
</tr>
<tr>
<td>English</td>
<td>546865726520 ...</td>
<td>There is a major center of economic...</td>
</tr>
</tbody>
</table>

Example encodings of a Linux32 Bind Shell
Basic components of an attack

- a NOP sled
- shellcode
- one or more pointers to shellcode

Defensive approaches

- Prevention of writing vulnerable code. Automatic bound checking for buffers
- Prevention of execution of injected code. Instruction set randomization
- Content-based input validation. Network traffic monitoring
Let the encoding begin!!!

"Shake Shake Shake!"

push %ebx; push "ake";
push %ebx; push "ake";
push %ebx; push "ake"
"Shake Shake Shake Shake!"

push %ebx; push "ake";
push %ebx; push "ake";
push %ebx; push "ake!";}
Let the encoding begin!!!

"Shake Shake Shake!"

push %ebx; push "ake";
push %ebx; push "ake";
push %ebx; push "ake";

<table>
<thead>
<tr>
<th>ASCII</th>
<th>HEX</th>
<th>ASSEMBLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>41</td>
<td>inc byte</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
<td>inc byte</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
<td>inc byte</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII</th>
<th>HEX</th>
<th>ASSEMBLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>09</td>
<td>dec byte</td>
</tr>
<tr>
<td>41</td>
<td>09</td>
<td>dec byte</td>
</tr>
<tr>
<td>42</td>
<td>09</td>
<td>dec byte</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STACK MANIPULATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>q</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>u</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>w</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII</th>
<th>HEX</th>
<th>ASSEMBLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>61</td>
<td>pop a</td>
</tr>
<tr>
<td>b</td>
<td>62</td>
<td>pop b</td>
</tr>
<tr>
<td>c</td>
<td>63</td>
<td>pop c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JUMPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>q</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>u</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>w</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>
How do we do it?

- Use an n-gram language model
- Deduce probability of word sequences
- Each instruction in decoder assigned a numerical value
- Compare candidate strings to number of instructions it produces the same effect of
- If first one, score is 1, if first two, score is 2 and so on
The Decoder

Mask it to sneak attack!!!

Self modification

Why?

There's no native support in English compatible instructions for loops. Gives away the decoder.

How?

Decoder would be proportional to length of encoded payload

Initialization

Expand working instruction set. More RIS2 instructions available

Decoder

Encoded payload
Self modification

Why?
There's no native support in English compatible instructions for loops.
Gives away the decoder.

How?
Decoder would be proportional to length of encoded payload
Attack

Initialization
Expands working instruction set
More x86 instructions available

Decoder

Encoded payload
Initialization

Expands working instruction set
More IA32 instructions available
Decoder

Guidelines

1. Use only English-compatible instructions.

2. Multiple and instructions (English-compatible) used to generate an add instruction (English-non-compatible).


Initialization

- Overwrite key matches registers and functions to non-compatible instructions.
- Copy distinguisher and add offset to mark encoded payload.
- Int. 31h above and below.
- Push virtual pointer onto stack --- with copy if on list --- pop.
- End / significant offset --- register --- argument multiple times.
- Push to stack --- push readable --- pop! This value is zero.
- Continue poping.

Decoding

- To facilitate hopping, non English compatible instructions are needed:
  - opcode for and = 0620 for space
  - 0x20

Initializing registers

Before decoder loop is executed, registers are written.

E.g. pepapi Opcode = 21.

Can be used to set registers df and to create ahl and hv in turn too.

Registers are populated by pushing values on the stack before popa executes.

Push equivalent to J (E hyped) and AH (hyped). Set jnunh2 pushes 0 and then "next".
Encoded payload
Decoder

Guidelines

1. Use only English-compatible instructions.

2. Multiple and instructions (English-compatible) used to generate an add instruction (English-compatible).

3. Favour push [eax] to push [edi].

Initialization

- Convert two registers and function in non-compatible instruction.
- Copy add-back pointer on each offset, to each encoded symbol.
- Copy the other and transfer.
- Pure virtual pointer onto stack with top two bits on top.
- Pop the significant offset and register to increment multiple times.
- Pop to execute each subword and pop into value to access restored pointer.

Unpacking the decoder

To facilitate copying, non-English-compatible instructions are needed:

- opcodes for and - 1001 for space
- Ox20

Initializing registers

Before decoder loop is executed, registers are written:

E.g. popa op code = 2
- can be used to set registers off and to create and in turn load.
- Registers are populated by pushing values on the stack before popa execute.
- Push equivalent to 3; (top), and 16 (higher).
- So, ‘jump’ pushes 0 and then ‘move’.

Decoding

<table>
<thead>
<tr>
<th>Decoder Value</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 2 pointers   |
| current real position via link offset from cell |
| output position for next decoded byte |

[Prezi Image]
Guidelines

1. Use only English-compatible instructions.

2. Multiple and instructions (English-compatible) used to generate an add instruction (English-non-compatible).

3. Favour push %eax ("P") to push %ecx ("Q").
Start
Write a decoder that is capable of encoding generic payloads using only English-compatible instructions.
Language Model Generation

Generate and train a natural language model with a large and diverse corpus of English text.
Using Viterbi search, traverse the language model, executing and scoring each candidate decoder.
Encode Target Shellcode

xor $eax, $eax
push $eax

Continue to traverse the language model, encoding the target shellcode as English. Upon delivery, this code will be decoded and executed.
Decoder

Guidelines

1. Use only English-compatible instructions.
2. Multiple and instructions (English-compatible) used to generate an add instruction (English-compatible).
3. Favour push 0x10 to push 0x00.

Initialization

- Overwrite: key register is read:
- push address pointer to add offset to stack
- push `for` and `while`

- Push virtual pointer onto stack:
- Add push by one byte
- push
- Last 5 significant bits
- push
- push to stack
- push index/pointer
- pop
- push index/pointer

Decoding

2 pointers

2. current
3. output position via
4. offset from index
5. output position for next decoded
6. byte
Initialization

- Overwrites key machine registers and patches in non-compatible instructions.
- Copy shellcode pointer and add offset to reach encoded payload.
- Inc? Too slow and tedious.
- Push shellcode pointer onto stack --> shift %esp by one byte --> pop --> last 3 significant bytes --> register --> increment multiple times --> push to stack --> stack realigned!!! --> pop this value to address encoded payload
Unpacking the decoder

To facilitate looping, non English compatible instructions are needed.

opcode for and = ASCII for space

0x20
Decoding

2 pointers
%esi - current
read position via
lods
offset from %edi
- output position
for next decoded
byte
Initializing registers

Before decoder loop is executed, registers are written.

Eg: popa's Opcode = 'a'.

Can be used to set registers of and to create add and in turn lods.

Registers are populated by pushing values on the stack before popa executes.
Push equivalent to "j" (1 byte) and "h" (4 bytes). So, "johnboat" pushes "0" and then "nboa".
Automatic Generation

3 types of instructions

- English compatible. No effect. Eg: NOP.
- Affect machine state. No adverse effects.
- Affect machine state adversely, but as a sum are benign.
3 types of instructions

• English compatible. No effect Eg: NOP.
• Affect machine state. No adverse effects.
• Affect machine state adversely but as a sum are benign.
a smoothed n-gram model. Length is 5.3.
ws will follow w1w2w3w4w5 only if w2w3w4w5 ws is in the training corpus.
Language model

Corpus

- Over 15,000 Wikipedia articles
- 37,000 books from Project Gutenberg
Corpus
Over 15000 wikipedia articles
27000 books from Project Gutenberg
a smoothed n-gram model. Length is 5^3. w5 will follow w1w2w3w4 only if w2w3w4w5 is in the training corpus
<table>
<thead>
<tr>
<th>ASSEMBLY</th>
<th>OPCODE</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>push %esp</td>
<td>54</td>
<td>68 65726520</td>
</tr>
<tr>
<td>push $20657265</td>
<td>6A 6F</td>
<td>6973 20 61206D61</td>
</tr>
<tr>
<td>imul %esi,20(%ebx),$616D2061</td>
<td>72 20</td>
<td>There is a major</td>
</tr>
<tr>
<td>push $6F</td>
<td>68 20617320</td>
<td>h as Star</td>
</tr>
<tr>
<td>jb short $22</td>
<td>74 61</td>
<td>Show. The form</td>
</tr>
<tr>
<td>push $20736120</td>
<td>72 20</td>
<td>States Dru</td>
</tr>
<tr>
<td>push %ebx</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>je short $63</td>
<td>68 6F772E20</td>
<td></td>
</tr>
<tr>
<td>jb short $22</td>
<td>72 6D</td>
<td></td>
</tr>
<tr>
<td>push %ebx</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>push $202E776F</td>
<td>74 61</td>
<td></td>
</tr>
<tr>
<td>push %esp</td>
<td>68 65206666F</td>
<td></td>
</tr>
<tr>
<td>jb short $6F</td>
<td>72 75</td>
<td></td>
</tr>
<tr>
<td>popad</td>
<td>61</td>
<td></td>
</tr>
</tbody>
</table>

There is a major center of economic activity, such as Star Trek, including The Ed Sullivan. Show. The former Soviet Union, International organization participation. Asian Development Bank, established in the United States Drug Enforcement Administration, and the Palestinian territories, the International Telecommunication Union, the first ma...
Conclusion

With great power comes great responsibility!!!

~Uncle Ben