Packet Inspection Praktikum

Martin Brunner, Jonas Pfoh
martin.brunner@sit.fraunhofer.de;pfoh@sec.in.tum.de

IT Security Lab
Technische Universität München
Munich, Germany

Sept. 6, 2010 - Sept. 24, 2010
Outline

1. Introduction
2. Buffer Overflows
3. Shellcode
4. Shellcode Detection
5. Further Information
Welcome (part 2)

**Goal:** (or what are we doing here)
- Some background information
- Give you an idea of existing methods for shellcode detection

You will learn:
1. What shellcode is and how it works
   - Exploiting process
   - Basics of buffer overflows
2. How shellcode can be recognized
Welcome (part 2)

**Goal:** (or what are we doing here)

- Some background information
- Give you an idea of existing methods for shellcode detection

You will learn:

1. What shellcode is and how it works
   - Exploiting process
   - Basics of buffer overflows
2. How shellcode can be recognized
What is shellcode 1

Example of a ('Shell spawning') Shellcode 1

\texttt{\textbackslash x31\textbackslash xc0\textbackslash xb0\textbackslash x46\textbackslash x31\textbackslash xdb\textbackslash x31\textbackslash xc9\textbackslash xcd\textbackslash x80}
\texttt{\textbackslash xeb\textbackslash x16\textbackslash x5b\textbackslash x31\textbackslash xc0\textbackslash x88\textbackslash x43\textbackslash x07\textbackslash x89\textbackslash x5b}
\texttt{\textbackslash x08\textbackslash x89\textbackslash x43\textbackslash x0c\textbackslash xb0\textbackslash x0b\textbackslash x8d\textbackslash x4b\textbackslash x08\textbackslash x8d}
\texttt{\textbackslash x53\textbackslash x0c\textbackslash xcd\textbackslash x80\textbackslash xe8\textbackslash xe5\textbackslash xff\textbackslash xff\textbackslash xff\textbackslash x2f}
\texttt{\textbackslash x62\textbackslash x69\textbackslash x6e\textbackslash x2f\textbackslash x73\textbackslash x68}

\footnote{taken from [1] p.40}
What is shellcode I

Example of a ('Shell spawning') Shellcode \(^1\)

\[ \text{x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80} \]
\[ \text{\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b} \]
\[ \text{\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d} \]
\[ \text{\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f} \]
\[ \text{\x62\x69\x6e\x2f\x73\x68} \]

\(^1\)taken from [1] p.40
What is shellcode I

Example of a ('Shell spawning') Shellcode

\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\n\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\n\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\n\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\n\x62\x69\x6e\x2f\x73\x68

1taken from [1] p.40
What is shellcode II

What is it used for (why do we want to detect it)?

Typically used as 'payload' within the process of exploiting a software vulnerability. Example buffer overflow:

- Injection vector: techniques used for directed overfilling of a buffer thus manipulating the program flow
- Payload: the injected program code that shall be executed
  (injection vector: cruise missille - payload: warhead)

Usage:

- Malicious: e.g., Malware
- Benign: e.g., Pentesting
What is shellcode II

What is it used for (why do we want to detect it)?

Typically used as 'payload' within the process of exploiting a software vulnerability. Example buffer overflow:

- Injection vector: techniques used for directed overfilling of a buffer thus manipulating the program flow
- Payload: the injected program code that shall be executed
  (injection vector: cruise missile - payload: warhead)

Usage:

- Malicious: e.g., Malware
- Benign: e.g., Pentesting
Background 1

**x86 architecture (IA-32)**

- Refers to both, the architecture and the instruction set of Intel 80386 compatible CPUs
- 32 Bit register (4 Bytes)
- Little endian format
  - Address register (ESI, EDI, ESP, EBP, EIP)
  - Data register (EAX, EBX, ECX, EDX)
    - 16-Bit fragments (AX, BX, CX, DX)
    - 8-Bit fragments (AH, AL, BH, BL, CH, CL, DH, DL)
## x86 architecture: register overview

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX, AX, AH, AL</td>
<td>Accumulator</td>
<td>arithmetic operations, input/output-operations, assignment of systemcall (includes also its return value)</td>
</tr>
<tr>
<td>EBX, BX, BH, BL</td>
<td>Base Register</td>
<td>indirect memory addressing, stores first argument of a syscall</td>
</tr>
<tr>
<td>ECX, CX, CH, CL</td>
<td>Counter</td>
<td>e.g., for loops, stores second argument of a syscall</td>
</tr>
<tr>
<td>EDX, DX, DH, DL</td>
<td>Data Register</td>
<td>stores addresses of variables and third argument of a syscall</td>
</tr>
<tr>
<td>ESI, EDI</td>
<td>Source Address Destination Address</td>
<td>used for long data-chains (e.g., arrays)</td>
</tr>
<tr>
<td>ESP</td>
<td>Stack Pointer</td>
<td>contains address on top of stack</td>
</tr>
<tr>
<td>EBP</td>
<td>Base Pointer</td>
<td>contains address of stack-bottom, points to local variables residing in the current stack-frame</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction Pointer</td>
<td>contains address of the instruction which gets executed next</td>
</tr>
</tbody>
</table>
Excursus: buffer overflows

**Principle:** unchecked (user) input gets copied to a buffer with a fixed length

Basic types of buffer overflows:

- Stack-based overflow
- Heap-based overflow
- Further: Off-by-one overflow, Frame Pointer overflow, BSS overflow
### Example: stack-based buffer overflow

**Important:** understand the memory layout

On IA-32: 32bit addr. space (4 GB, 0x00000000-0xFFFFFFF)

<table>
<thead>
<tr>
<th>Memory Area</th>
<th>Content / Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>programm environment (name, arguments, environment variables), Userstack (function call parameter, return address, basepointer, local variables of the corresponding called function) <em>Stack grows in direction of Heap</em></td>
</tr>
<tr>
<td>Heap</td>
<td>dynamic data structures (malloc / free), <em>Heap grows in direction of Stack</em></td>
</tr>
<tr>
<td>BSS Segment</td>
<td>Not (yet) initialized global variables</td>
</tr>
<tr>
<td>Data Segment</td>
<td>initialized global variables</td>
</tr>
<tr>
<td>Text Segment</td>
<td>compile source code</td>
</tr>
</tbody>
</table>
Program processes user input and writes it into buffer without validation: i.e., strcpy();

Return address gets overwritten (uncontrolled/controlled), since Stack grows towards Heap

CPU executes instruction stored on address of the value of return address:

When overwritten with arbitrary (invalid) data: segmentation fault (local DoS)

If attacker’s goal was to crash the application he succeeded

BUT: the attacker can also use the vulnerability to inject code
Exploiting stack-smashing buffer overflows
Memory 'behind' gets overwritten - 'behind' resides EBP, EIP, arguments, local variables. Once EIP gets overwritten:
- At the end of function the EIP stored on the Stack gets copied to the 'real' EIP
- Programm continues where EIP points to
- If EIP points to Stack: code on the Stack is executed

Payload:
- NOPs: \x90 (count: buffer size - shellcode-length -4)
- Shellcodes
- Return address (determination not covered here)
  - inside NOP-sled
  - fixed (was 0xbfffffffff for Linux/BSD)
  - find out using gdb

Further methods (not discussed here): return-into-libc, storing shellcode in environment variables, etc.
Shellcode I

What is shellcode

- Compact sequence of instructions (typically in order to start a shell, thus shellcode)

- Notation: hex / escaped (\x90) for use in exploits

- The hex values represent the corresponding opcodes

- Shellcode is 'non-system' code (since it gets injected) and must fulfill several requirements:
  - Size matters: shellcode is limited by the buffer length
  - Must not contain specific characters - on IA-32 e.g.,
    - 0x00 - string termination
    - 0x0a - linefeed
    - 0x0d - carriage return
    - 0x0b, 0x0c - quits string-processing via sscanf()

- Shellcode is typically exactly adapted to the target platform in order gain maximum functionality while requiring minimal space

\(^2\)ASM instructions for the target platform
Types of shellcode

- 'Normal' (shell-spawning) shellcode: opens a shell (e.g., /bin/sh) - used for local exploits, especially interesting for programs with SETUID-Bit (local privilege escalation!)
- Bindshell: opens a port and attaches a shell to it
- Reverseshell: connects to a remote port and attaches a shell to it
- ...

...
Shellcode III

Types of shellcode (cont.) - enhancements

- Encoding (XOR, polymorphic, alphanumeric):
  - Obfuscation/evasion of IPS
  - Avoiding bad characters (such as 0x00)

- Staged loading shellcode: complex functionality requires 2-staged attack - e.g., egghunt-concept form Windows targets

- 'Tolerant' shellcodes
  - Overall architectures
    - Valid opcode for 2 architectures gets used
    - Program flow control depending on architecture
    - Shellcode grows
  - OS-independent:
    - Use of existing shellcode
    - Syscalls available in every OS but have different meaning
    - Based on return values: decision (during runtime) on which OS the shellcode was run
Shellcode IV

Getting / generating shellcode

Basic procedure (Linux):

- C source
- Typically execve or execl ("/bin/sh","sh",0);
- ASM program (gcc -o prog -static -ggdb prog.c)
- Conversion in list of executable opcodes
  - Manually in ASM code
  - Trying ASM calls in C
  - Testing: direct execution of opcodes
- Avoid 0x00
- INT 0x80 - causes programmed exception and calls the kernel system call routine
- Syscall reference: /usr/include/asm/unistd.h

Download existing (ready-to-use) shellcode
Use tools (s. below)
Shellcode detection 1

Exemplary challenges

- Alphanumeric shellcode
  - Uses ASCII-printable chars only (0x33-0x7e)
  - Usage of INC and DEC for registers instead of NOP (0x90)

- Polymorphic Shellcode
  - Used for evading IPS capable of detecting not only NOPs but also well-known shellcode-signatures
  - Shellcode is encrypted (e.g., via XOR)
  - Loader code decrypts and builds the ’real’ shellcode during runtime
  - Every time the loader code gets newly generated and a new key is used for encoding the shellcode. Thus pattern/signature matching approaches (as performed by traditional IPS and anti-virus software) fails

- Big research area!
Existing approaches (digest!)

- Static (IDA pro, OllyDbg)
- Dynamic (using sandboxes and logging traps/syscalls e.g., Spector, Norman Sandbox, CW-Sandbox, Ether, etc.)

Look for shellcode-typical patterns e.g.,
- NOP sleds (e.g., STRIDE)
- C-functions used for filtering/blocking characters, e.g., strspn() or strcspn()

- Snort-IDS (static signatures)
- Bro-IDS (supports additionally regex)
Existing approaches (digest!) cont.

- **Emulation**
  - LibEmu \(^3\): uses so-called GetPC heuristic
    - GetPC- Get Program Counter: sequence of ASM instructions in malicious code
    - Necessary to identify the value of the program counter and thus the position in memory
    - LibEmu looks for corresponding (suspicious) combinations of ASM instructions in order to detect shellcode
  - Emulation and heuristics for detecting polymorphic shellcode e.g., [10]

- Identify ranges of possible return addresses for existing buffer-overflow vulnerabilities (e.g., Buttercup, also included in Snort)

---

\(^3\)http://libemu.carnivore.it/
Useful Shellcode Tools

Shellcode generators

1. Metasploit Meterpreter - www.metasploit.com
2. Hellkit -
   packetstorm.linuxsecurity.com/groups/teso/hellkit-1.2.tar.gz
3. ShellForge - www.secdev.org/python/shellforge.py
4. HOON - felinemenace.org/nd/HOON.tar.bz2
5. InlineEgg - oss.coresecurity.com/projects/inlineegg.html
6. ADMmutate - www.ktwo.ca/ADMmutate-0.8.4.tar.gz

Utils

1. nasm (nasm –f elf code.asm)
2. objdump (objdump –d code.o –h dump)
3. odfhex (odfhex dump) - converts shellcode (from objdump –d) in escaped hexcode
   http://www.acm.uiuc.edu/sigmil/talks/shellcode/odfhex.cpp


5 AlephOne: Smashing the Stack for Fun and Profit. Phrack Magazine, 49:14, November 1996


