Basic exploitation techniques

20180118
Outline

A primer on x86 assembly

Memory segments

Stack-based buffer overflows

Heap-based overflow

Format strings
A primer on x86 assembly
Introduction

Verily, when the developer herds understand the tools that drive them to their cubiced pastures every day, then shall the Oday be depleted — but not before.
– Pastor Manul Laphroaig
It’s a trap!

- ≈ 1000 instructions . . .
- No time to know them all :-)

This overview is meant as a first help

Multiple syntaxes

- AT&T
- Intel
## Basics

### In general

Mnemonics accept from 0 to 3 arguments.

2 arguments mnemonics are of the form (Intel syntax)

\[ m \text{ dst, src} \]

which roughly means

\[ \text{dst} \leftarrow \text{dst} \odot \text{src} \]

where \( \odot \) is the semantics of \( m \)
Endianness

\[
x = 0x\text{deadbeef}
\]

<table>
<thead>
<tr>
<th>byte address</th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte content (big-endian)</td>
<td>0xde</td>
<td>0xad</td>
<td>0xbe</td>
<td>0xef</td>
</tr>
<tr>
<td>byte content (litte-endian)</td>
<td>0xef</td>
<td>0xbe</td>
<td>0xad</td>
<td>0xde</td>
</tr>
</tbody>
</table>

- Big endian (PowerPC, Sparc, 68000)
- Little endian (Intel, AMD, ARM (usually), RISC-V)
Resources

- Cheat sheet
- Opcode and Instruction Reference
- Intel full instruction set reference
Basic registers (16/32/64 bits)

<table>
<thead>
<tr>
<th>64</th>
<th>32</th>
<th>16</th>
<th>name (8080) / use</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>e</td>
<td>ax</td>
<td>accumulator</td>
</tr>
<tr>
<td>r</td>
<td>e</td>
<td>bx</td>
<td>base address</td>
</tr>
<tr>
<td>r</td>
<td>e</td>
<td>cx</td>
<td>count</td>
</tr>
<tr>
<td>r</td>
<td>e</td>
<td>dx</td>
<td>data</td>
</tr>
<tr>
<td>r</td>
<td>e</td>
<td>di</td>
<td>source index</td>
</tr>
<tr>
<td>r</td>
<td>e</td>
<td>si</td>
<td>destination index</td>
</tr>
<tr>
<td>r</td>
<td>e</td>
<td>bp</td>
<td>base pointer</td>
</tr>
<tr>
<td>r</td>
<td>e</td>
<td>sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>r</td>
<td>e</td>
<td>ip</td>
<td>instruction pointer</td>
</tr>
</tbody>
</table>

- esp (e = extended) is the 32 bits stack pointer
- rsp (r = register) is the 64 bits one
Add extended general purpose registers r8-15

- \(r7^d\) accesses the lower 32 bits of \(r7\);
- \(r7^w\) the lower 16 bits;
- \(r7^b\) its lower 8 bits.
The full story
Register flags (partial)

- **of**: overflow flag
- **cf**: carry flag
- **zf**: zero flag
- **sf**: sign flag
- **df**: direction flag
- **pf**: parity flag
- **af**: adjust flag
Signed vs unsigned

At machine-level, every value is a bitvector, which can be seen through different lenses:

- unsigned value
- signed value
- float (will not talk about it)
Move

mov dst, src  dst := src
xchg o1, o2  tmp := o1; o1 := o2; o2 := tmp
All 4 arithmetic operations are present

<table>
<thead>
<tr>
<th>Operation</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add src, dst</td>
<td>dst ← dst + src</td>
</tr>
<tr>
<td>sub src, dst</td>
<td>dst ← dst - src</td>
</tr>
</tbody>
</table>
| div src | t64 ← edx @ eax  
eax ← tmp / src  
edx ← tmp % src |
| mul src | t64 ← eax * src  
edx ← tmp{32,63}  
eax ← tmp{0,31} |
Arithmetic

**inc dst** \[\text{dst} \leftarrow \text{dst} + 1\]

**dec dst** \[\text{dst} \leftarrow \text{dst} - 1\]

**sal/sar dst, src** arithmetic shift left / right

**Sign preservation**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mov ax, 0xff00</td>
<td>unsigned: 65280, signed: -256</td>
</tr>
<tr>
<td></td>
<td># ax=1111.1111.0000.0000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>sal ax, 2</td>
<td>unsigned: 64512, signed: -1024</td>
</tr>
<tr>
<td></td>
<td># ax=1111.1100.0000.0000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sar ax, 5</td>
<td>unsigned: 65504, signed: -32</td>
</tr>
<tr>
<td></td>
<td># ax=1111.1111.1110.0000</td>
<td></td>
</tr>
</tbody>
</table>
## Basic logical operators

### Basic semantics

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>dst, src</td>
<td>dst ← dst &amp; src</td>
</tr>
<tr>
<td>or</td>
<td>dst, src</td>
<td>dst ← dst</td>
</tr>
<tr>
<td>xor</td>
<td>dst, src</td>
<td>dst ← dst ^ src</td>
</tr>
<tr>
<td>not</td>
<td>dst</td>
<td>dst ← ~dst</td>
</tr>
</tbody>
</table>

### Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor ax, ax</td>
<td># ax = 0x0000</td>
</tr>
<tr>
<td>not ax</td>
<td># ax = 0xffff</td>
</tr>
<tr>
<td>mov bx, 0x5500</td>
<td># bx = 0x5500</td>
</tr>
<tr>
<td>xor ax, bx</td>
<td># ax = 0xbbff</td>
</tr>
</tbody>
</table>
## Logical shifts

### Shift

- **shl dst, src**  
  shift left
- **shr dst, src**  
  shift right

### Example

```assembly
1 mov ax, 0xff00  # unsigned: 65280, signed: -256
  # ax=1111.1111.0000.0000
2 shl ax, 2       # unsigned: 64512, signed: -1024
  # ax=1111.1100.0000.0000
3 shr ax, 5       # unsigned: 2016, signed: 2016
  # ax=0000.0111.1110.0000
```
Comparison and test instructions

<table>
<thead>
<tr>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp ( \text{dst} \ \text{src} ) : set condition according to ( \text{dst} - \text{src} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>test ( \text{dst} \ \text{src} ) : set condition according to ( \text{dst}&amp;\text{src} )</td>
</tr>
</tbody>
</table>
## Stack manipulation

<table>
<thead>
<tr>
<th>Push</th>
<th>Pop</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>push src dec sp; @[sp] := src</code></td>
<td><code>pop src src := @[sp]; inc sp</code></td>
</tr>
</tbody>
</table>
### Misc

<table>
<thead>
<tr>
<th>Nop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does nothing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lea (load effective address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea dst, [src] dst := src</td>
</tr>
<tr>
<td>mov dst, [src] dst := @[src]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>int n  runs interrupt number n</td>
</tr>
</tbody>
</table>

19
# Unconditional jump instructions

<table>
<thead>
<tr>
<th>Call</th>
<th>Jmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>call address</td>
<td>jmp *op</td>
</tr>
<tr>
<td>call *op</td>
<td>jmp address</td>
</tr>
<tr>
<td>• pushes eip</td>
<td>nothing else</td>
</tr>
</tbody>
</table>

- push eip

20
### Extra jump

#### Leave
- \( \text{esp} := \text{ebp} \); \( \text{ebp} := \text{pop}() \);

#### Ret
\[
\text{esp} := \text{esp} + 4; \quad \text{eip} := \@[\text{esp} - 4];
\]
Unsigned jumps

<table>
<thead>
<tr>
<th>jump</th>
<th>if</th>
<th>n version</th>
<th>e version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ja</td>
<td>above</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jb</td>
<td>below</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jc</td>
<td>carry</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>

Reading

ja has n and e versions, means that mnemonics

- jna (not above),
- jae (above or equal),
- jnae (not above or equal)

exist as well
## Signed jumps

<table>
<thead>
<tr>
<th>jump type</th>
<th>if</th>
<th>n version</th>
<th>e version</th>
</tr>
</thead>
<tbody>
<tr>
<td>jg</td>
<td>greater</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jl</td>
<td>lower</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>jo</td>
<td>overflow</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>js</td>
<td>if sign</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>
The **addressing mode** determines, for an instruction that accesses a memory location, how the address for the memory location is specified.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>mov ax, 16h</td>
</tr>
<tr>
<td>Direct</td>
<td>mov ax, [1000h]</td>
</tr>
<tr>
<td>Register Direct</td>
<td>mov bx, ax</td>
</tr>
<tr>
<td>Register Indirect (indexed)</td>
<td>mov ax, [di]</td>
</tr>
<tr>
<td>Based Indexed Addressing</td>
<td>mov ax, [bx + di]</td>
</tr>
<tr>
<td>Based Indexex Disp.</td>
<td>mov eax, [ebx + edi + 2]</td>
</tr>
</tbody>
</table>
The semantics of instructions is complex.
Instructions have side effects

```
// 04 16 / add al, 0x16
0: res8 := (eax(32){0,7} + 22(8))
1: OF := ((eax(32){0,7}{7} = 22(8){7}) &
    (eax(32){0,7}{7} != res8(8){7}))
2: SF := (res8(8) <s 0(8))
3: ZF := (res8(8) = 0(8))
4: AF := ((extu eax(32){0,7}{0,7} 9) + 22(9)){8}
5: PF := !
    (((((((res8(8){0} ^ res8(8){1}) ^ res8(8){2}) ^
      res8(8){3}) ^ res8(8){4}) ^ res8(8){5}) ^
    res8(8){6}) ^ res8(8){7}))
6: CF := ((extu eax(32){0,7} 9) + 22(9)){8}
7: eax{0, 7} := res8(8)
```
Real behavior of conditions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Flag</th>
<th>cmp x y</th>
<th>sub x y</th>
<th>test x y</th>
</tr>
</thead>
<tbody>
<tr>
<td>ja</td>
<td>¬ CF ∧ ¬ ZF</td>
<td>x &gt;_ y</td>
<td>x’ ≠ 0</td>
<td>x&amp;y ≠ 0</td>
</tr>
<tr>
<td>jnae</td>
<td>CF</td>
<td>x &lt;_ y</td>
<td>x’ ≠ 0</td>
<td>⊥</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>x = y</td>
<td>x’ = 0</td>
<td>x&amp;y = 0</td>
</tr>
<tr>
<td>jge</td>
<td>OF = SF</td>
<td>x ≥ y</td>
<td>T</td>
<td>x ≥ 0 ∨ y ≥ 0</td>
</tr>
<tr>
<td>jle</td>
<td>ZF ∨ OF ≠ SF</td>
<td>x ≤ y</td>
<td>T</td>
<td>x&amp;y = 0 ∨ (x &lt; 0 ∧ y &lt; 0)</td>
</tr>
</tbody>
</table>
Shift left

The OF flag is affected **only on 1-bit shifts**. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

The OF flag is affected only for 1-bit shifts (see "Description" above); otherwise, it is **undefined**.
Memory segments
A compiled program has 5 segments:

1. code (text)
2. stack
3. data segments
   3.1 data
   3.2 bss
   3.3 heap
1. Read instruction $i \circ eip$
2. Add byte length of $i$ to eip
3. Execute $i$
4. Goto 1
Graphically speaking

- Stack
- The hole
- Heap
- BSS
- Data
- Text
- Function, locals
- The break
- Malloc, free
- Globals
Text segment

- The text segment (aka code segment) is where the code resides.
- It is not writable. Any attempt to write to it will kill the program.
- As it is ro, it can be shared among processes.
- It has a fixed size
Data & bss segments

- The **data segment** is filled with **initialized** global and static variables.
- The **bss segment** contains the **uninitialized** ones. It is zeroed on program startup.
- The segments are (of course) writable.
- They have a **fixed size**

---

stack

↓

the hole

↓

heap

---

bss

data

text
Heap segment

- The heap segment is directly controlled by the programmer.
- Blocks can be allocated or freed and used for anything.
- It is writable.
- It can grow larger, towards higher memory addresses – or smaller, on need.
The stack segment is a temporary scratch pad for functions.

Since eip changes on function calls, the stack is used to remember the previous state (return address, calling function base, arguments, \ldots).

It is writable.

It can grow larger, towards lower memory addresses – w.r.t to function calls.
void test_function(int a, int b, int c, int d) {
    int flag;
    char buffer[10];
    flag = 31337;
    buffer[0] = 'A';
}

int main() {
    test_function(1, 2, 3, 4);
}
Stack-based buffer overflows
In C, the programmer is responsible for data integrity. This means there are no guards to ensure data is freed, or that the contents of a variable fits into memory. This exposes memory leaks and buffer overflows.
Reminder: stack layout for x86

Code
f: ...
call g
...

Data
val₁
val₂

stack frame f

return address f
saved frame pointer f

locals f

arguments g

return address g
saved frame pointer g

pointer to data

buffer

stack frame g

locals g
Vulnerability reason

- When an array \( a \) is declared in C, space is reserved for it.
- \( a \) will be manipulated through offsets from its base pointer.
- At run-time, no information about the array size is present.
- Thus, it is allowed to copy data beyond the end of \( a \)
A rich history

1972  First document attack
1988  Morris worm
1995  NCSA http 1.3
1996  Smashing the STack
Basic exploitation

Code
f: ...
call g ...

Data
val1
val2

return address f
saved frame pointer f

locals f

arguments g

return address g
saved frame pointer g

pointer to data

injected code
Frame pointer overwriting

Code

f: ...
call g
...

Data

val₁
val₂

return address f
saved frame pointer f

locals f

arguments g

return address g
saved frame pointer g

pointer to data

return address f
saved frame pointer f

injected code
Indirect pointer overwriting

Code
f: ...
call g
...

Data
val_1
val_2

stack frame f

return address f
saved frame pointer f
locals f
arguments g
return address g
saved frame pointer g
pointer to data
injected code

stack frame g

stack frame g

locals g

locals g
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];
    strcpy(password_buffer, password);
    if (strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if (strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if (argc < 2) { printf("Usage: %s <password>\n", argv[0]); exit(0); }
    if (check_authentication(argv[1])) printf("\nAccess Granted.\n");
    else printf("\nAccess Denied.\n");
}
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_authentication(char *password) {
    char password_buffer[16];
    int auth_flag = 0;
    strcpy(password_buffer, password);
    if (strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if (strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if (argc < 2) { printf("Usage: %s <password>\n", argv[0]); exit(0); }
    if (check_authentication(argv[1])) printf("Access Granted.\n");
    else printf("Access Denied.\n");
}

## Constraints

### Needs
- Hardware willing to execute data as code
- No null bytes

### Variants
- Frame pointer corruption
- Causing an exception to execute a specific function pointer
Heap-based overflow
Heap memory is dynamically allocated at runtime.

Arrays on the heap overflow just as well as those on the stack.

**Warning**

The heap grows towards higher addresses instead of lower addresses.

This is the opposite of the stack.
Basic exploitation

Overwriting heap-based function pointers located after the buffer

Overwriting virtual function pointer

1998  IE4 Heap overflow
2002  Slapper worm
CVE-2007-1365  OpenBSD 2^{nd} remote exploits in 10 years
CVE-2017-11779  Windows DNS client
Overwriting heap-based function pointers

typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*, char*);
} vulnerable;

int is_file_foobar_using_heap(vulnerable* s, char* one, char* two)
{
    strcpy(s->buff, one);
    strcat(s->buff, two);
    return s->cmp(s->buff, "foobar");
}
Constraints

- Ability to determine the address of heap
- If string-based, no null-bytes

Variants

- Corrupt pointers in other (adjacent) data structures
- Corrupt heap metadata
Format strings
About format strings vulnerabilities

They were the ‘spork‘ of exploitation. ASLR? PIE? NX Stack/Heap? No problem, fmt had you covered.
Vulnerability

Format functions are variadic.

```c
int printf(const char *format, ...);
```

**How it works**

- The format string is copied to the output unless `%` is encountered.
- Then the format specifier will manipulate the output.
- When an argument is required, it is expected to be on the stack.
Caveat

And so ..

If an attacker is able to specify the format string, it is now able to control what the function pops from the stack and can make the program write to arbitrary memory locations.

CVEs

<table>
<thead>
<tr>
<th>Software</th>
<th>CVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zend</td>
<td>2015-8617</td>
</tr>
<tr>
<td>latex2rtf</td>
<td>2015-8106</td>
</tr>
<tr>
<td>VmWare 8x</td>
<td>2012-3569</td>
</tr>
<tr>
<td>WuFTPD (providing remote root since 1994)</td>
<td>2000</td>
</tr>
</tbody>
</table>
Good & Bad

**Good ✔**

```c
int f (char *user) {
    printf("%s", user);
}
```

**Bad ✗**

```c
int f (char *user) {
    printf(user);
}
```
Exploitation

Badly formatted format parameters can lead to:

- arbitrary memory read (data leak)
- arbitrary memory write
  - rewriting the .dtors section
  - overwriting the Global Offset Table (.got)
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char text[1024];
    static int test_val = 65;
    if (argc < 2) {
        printf("Usage: %s <text to print>\n", argv[0]);
        exit(0);
    }
    strcpy(text, argv[1]);
    printf("The right way to print user-controlled input:\n");
    printf("%s", text);
    printf("\nThe wrong way to print user-controlled input:\n");
    printf(text);
    // Debug output
    printf("\n[*] test_val @ 0x%08x = %d 0x%08x\n",
           &test_val, test_val, test_val);
    exit(0);
}
```
The %s format specifier can be used to read from arbitrary addresses

1 $ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
2 The right way to print user-controlled input:
3 AAAA%08x.%08x.%08x.%08x
4 The wrong way to print user-controlled input:
5 AAAAffffffc0.f7ffcfdd.565555c7.41414141
6 [*] test_val @ 0x56557028 = 65 0x00000041
$ ./fmt_vuln $(printf "\x28\x70\x55\x56")%08x.%08x.%08x.%s

The right way to print user-controlled input:
(pUV%08x.%08x.%08x.%s

The wrong way to print user-controlled input:
(pUVffffffc0.f7ffcfd4.565555c7.A

[*] test_val @ 0x56557028 = 65 0x00000041

65 is the ASCII value of 'a'
Writing to arbitrary memory

As %s, %n can be used to write to arbitrary addresses.

```plaintext
1 $ ./fmt_vuln $(printf "\x28\x70\x55\x56")%08x.%08x.%08x.%n
2 The right way to print user-controlled input:
3 (pUV%08x.%08x.%08x.%n
4 The wrong way to print user-controlled input:
5 (pUVffffffc0.f7ffcfd4.565555c7.
6 [*] test_val @ 0x56557028 = 31 0x0000001f
```
Play games

https://microcorruption.com
Questions ?