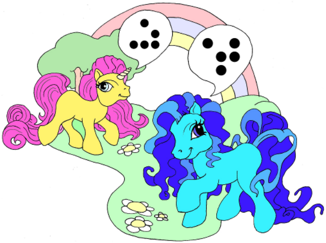
Zomg pwnies!



Morten Brøns-Pedersen
Department of Computer Science

Overview

Website: slamkode.dk

This is the stuff we'll go over today:

- Introduction.
- Reversing.
- Shellcode.
- Stack buffer overflow.
- Heap buffer ovorflow and Doug Lea's malloc-and-friends.
- Writing exploits.

Introduction

- What is our goal?
- What is Pwnies' teaching form?
- Which prerequisites should you have?
- What do we expect of you?
- What will you learn?

Send a team from DIKU to DEFCON CTF in 2011.

Introduction

- What is our goal?
- What is Pwnies' teaching form?
- Which prerequisites should you have?
- What do we expect of you?
- What will you learn?

We meet every Sunday as long as anyone is still motivated. A lecture in the morning and workshop in the afternoon. The first four lectures are already planned - after that *you* will do the talking.

Introduction

- What is our goal?
- What is Pwnies' teaching form?
- Which prerequisites should you have?
- What do we expect of you?
- What will you learn?

None. We'll start with the basic stuff and work our way from there. But you should have looked a chapters 0x100 and 0x200 in "Hacking: The Art of Exploitation" and installed IDA Pro for today.
A lot of the ground we cover in Pwnies is similar to the curriculum of Proactive Computer Security (block 4 in 2011).

Introduction

- What is our goal?
- What is Pwnies' teaching form?
- Which prerequisites should you have?
- What do we expect of you?
- What will you learn?

You should really try to be here every Sunday. Otherwise you will have a hard time. In addition to the lectures and workshops there will be some homework, both reading and exercises.

Introduction

- What is our goal?
- What is Pwnies' teaching form?
- Which prerequisites should you have?
- What do we expect of you?
- What will you learn?

Our main focus is binary exploitation. That means a *lot* of reversing. We really need people who can run HandRays as fast as a modern computer can run HexRays.

But we cover other topics too. The first four weeks focus on

Week 0 Binary exploitation (Morten).

Week 1 Forensics (Irfan).

Week 2 Crypto (Johan).

Week 3 Advanced reversing with IDA Pro (Jesper).

Binary exploitation

For most of the problems we'll look at today we're after code execution. In general that involves three steps:

- Get your (shell)code into memory.
- Get control of execution.
- Execute your (shell)code.

Often this is not a big problem. For today the only requirement is that your shellcode does not contain any NUL bytes.

Binary exploitation

For most of the problems we'll look at today we're after code execution. In general that involves three steps:

- Get your (shell)code into memory.
- Get control of execution.
- Execute your (shell)code.

This is where you need to find a bug in the target program. A typical example is to overflow a stack allocated buffer and overwrite the return address. We'll look into this later.

Binary exploitation

For most of the problems we'll look at today we're after code execution. In general that involves three steps:

- Get your (shell)code into memory.
- Get control of execution.
- Execute your (shell)code.

Most of the time you don't have a clue where in memory your shellcode is located. So it can be tricky to get it executed even though you have control over the instruction pointer.

Reversing (finding the bug)

It *is* possible to exploit a program without a clue of how it works (someone^{TM} should hold a lecture on fuzzing) but we will try to be a bit more ninja about it.

At DEFCON CTF you don't get the source code for the executables. So the only way to know how the program works is to reverse it.

Here at Pwnies we will use IDA Pro. IDA Pro is a Windows program, but it runs OK under wine. And it ships with a Linux debugger.

Focus

Today we will focus on

- Intel X86 32 bits architecture
- ELF executables.
- X86 assembler (Intel syntax).
- Linux.

This is a good starting point mainly because of the good documentation.

(Almost) general purpose registers

See "The Art of Picking Intel Registers" ([Swanson 03]).

EAX Accumulator. Results.

EBX Base register (data pointer). General purpose.

ECX Loop counter.

EDX Data register, I/O pointer. 64-bit extension of EAX.

ESI Source register.

EDI Destination register.

EBP Base pointer.

ESP Stack pointer.

Register addressing

Example: EAX is 32 bits, AX is the lower 16 bits of EAX, AH is the upper 8 bits of AX and AL is the lower 8 bits of AX.

Other registers

EIP Instruction pointer. Can't be controlled directly.

Eflags Indicate events. For example ZF (zero flag), SF (sign flag) and CF (carry flag). See "Intel Architecture Software Developer's Manual, Volume 1" ([Intel 99a]).

Segment registers

There are six: CS, DS, SS, ES, FS and GS.

Addresses are 48 bits: a 16 bit segment selector and a 32 bit offset. In reality segmentation is practically disabled. Code is data and data is code, pointers are 32 bits.

Segment FS is an exception, it might change from thread to thread.

Types

- Byte: 8 bits.
- Word: 2 bytes, 16 bits.
- Doubleword (or dword): 2 words, 32 bits.
- Quadword (or qword): 2 dwords, 64 bits.
- Doublequadword: 2 qwords, 128 bits.

Tools

IDA Pro The tool, hands down. We will use IDA Pro from day 0, so if you don't have it installed already: Get it! Week 3 will focus on using IDA Pro.

objdump and gdb OK tools if you just need something lightweight. Good for working over ssh, for example.

Intel X86 assembler

- Get a copy of "Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference" ([Intel 99b]).
- gdb uses AT&T syntax, we use Intel syntax.
- Little endian (least significant byte has lowest address).

Some OP codes

PUSH src Pushes it's operand onto the stack. Decreases ESP according to the operand. If ESP itself is pushed, it is the value of ESP *before* the operation that gets pushed.

POP dst Pops a value from the stack and stores it in the operand. Increases ESP accordingly.

MOV dst, src Moves the value of src to dst. The source operand can be a value, a register or a memory location (using the [reg1 + reg2 * imm1 + imm2] notation). The destination can be a register or a memory location.

ADD dst, src Adds src and dst and stores the result in dst.

SUB dst, src You guessed it.

Some OP codes

CMP op1, op2 Compares op1 to op2 by subtracting op2 from op1, and sets CF, OF, SF, ZF, AF and PF in Eflags accordingly. Normally used in conjunction with jump instructions.

JZ, JNZ, JL, JLE, JG, JGE Jump according to Eflags. If Eflags was set by CMP op1 op2, then the OP codes mean jump if op1 is equal to, not equal to, less than, less than or equal to, greater than or greater than or equal to op2 respectively. There are many other OP codes in the Jcc family.

JMP dst Jump to dst (register, memory, offset or absolute address).

CALL dst Push the address of the following OP code on the stack and jump to dst.

RET Pop the return address and jump to it.

Some OP codes

Look these up yourself:

LEA, PUSHA, POPA, PUSHF, POPF, CDQ, AND, OR, NOT, XOR, BSF, BSR, BSWAP, XCHG, MUL, DIV, NEG, NOP, INC, DEC, ENTER, LEAVE, LOOP, STOS, ROL, ROR, SHL, SHR, XADD, XLAT, and many more...

ELF and virtual memory

ELF is a format for storing executables, object files and shared libraries.

The ELF file describes how the memory should be laid out (what goes where), and how linking is to be done.

Our programs run in virtual memory meaning that every program "thinks" that is has its own (rather large) chunk of memory. Direct memory references is possible.

The stack

The stack grows down from higher addresses to lower addresses.

The top of the stack is pointed to by ESP. To allocate space on the stack *subtract* from ESP, and to free space on the stack *add* to it.

Remember ESP points to the last value pushed.

Function calls

Refer to [Erickson 08] for details.

The way a function call is done is the caller first pushes the arguments (right to left) and the return address onto the stack. Then the callee pushes EBP onto the stack and stores ESP in EBP. Local variables are allocated on the stack by subtracting from ESP.
After the function has executed ESP and EBP is brought back to normal.

Try and write a very small program and reverse it to see how it all works!

The heap

The heap is used for dynamically allocated memory. The heap is managed by a memory allocator. There are many implementations. Later today we will learn how to exploit Doug Lee's allocator (usually just called dlmalloc).

System calls

The program communicate with the operating system by system calls. System calls on Linux are particularly simple.

EAX holds the system call number, and EBX, ECX, EDX, ESI, EDI and EBP hold the arguments. When the registers are set up right interrupt 0x80 executes the system call. The OP code is INT 0x80.

See manpage syscalls(2).

Examples

If-statement

```
080483e4 <main>:
 80483e4: push   ebp
 80483e5: mov    ebp,esp
 80483e7: and    esp,0xfffffff0
 80483ea: sub    esp,0x10
 80483ed: cmp    DWORD [ebp+0x8],0x1
 80483f1: jg     80483ff <main+0x1b>
 80483f3: mov    DWORD [esp], 0x80484d0
 80483fa: call   8048318 <puts@plt>
 80483ff: mov    eax,0x0
 8048404: leave
 8048405: ret
```

Loop

```
080483b4 <main>:
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x70
 80483ba: mov    DWORD [ebp-0x4],0x0
 80483c1: jmp    80483df <main+0x2b>
 80483c3: mov    eax,[ebp-0x4]
 80483c6: mov    edx,[ebp+0xc]
 80483c9: add    edx,0x4
 80483cc: mov    ecx,[edx]
 80483ce: mov    edx,[ebp-0x4]
 80483d1: lea    edx,[ecx+edx]
 80483d4: movzx  BYTE edx,[edx]
 80483d7: mov    [ebp+eax-0x68],dl
 80483db: add    DWORD [ebp-0x4],0x1
 80483df: cmp    DWORD [ebp-0x4],0x63
 80483e3: jle    80483c3 <main+0xf>
 80483e5: mov    eax,0x0
 80483ea: leave
 80483eb: ret
```

Relative jumps

What does these OP codes do?

```
deadbeef: eb 00    jmp 0x0
deadbef1: eb fe    jmp -0x2
```

What about this one?

```
decafbad: e8 ff ff ff ff   call -0x1
decafbb3: c0 58            [jiberish]
```

Gcc stack protector

(AT&T syntax for the heck of it)

```
08048404 <main>:
 8048404: 55                    push   %ebp
 8048405: 89 e5                 mov    %esp,%ebp
 8048407: 83 e4 f0              and    $0xfffffff0,%esp
 804840a: 83 c4 80              add    $0xffffff80,%esp
 804840d: 8b 45 0c              mov    0xc(%ebp),%eax
 8048410: 89 44 24 0c           mov    %eax,0xc(%esp)
 8048414: 65 a1 14 00 00 00     mov    %gs:0x14,%eax
 804841a: 89 44 24 7c           mov    %eax,0x7c(%esp)
 804841e: 31 c0                 xor    %eax,%eax
 8048420: c7 44 24 14 00 00 00  movl   $0x0,0x14(%esp)
 ...
 8048456: 8b 54 24 7c           mov    0x7c(%esp),%edx
 804845a: 65 33 15 14 00 00 00  xor    %gs:0x14,%edx
 8048461: 74 05                 je     8048468 <main+0x64>
 8048463: e8 d8 fe ff ff        call   8048340 <__stack_chk_fail@plt>
 8048468: c9                    leave
 8048469: c3                    ret
```

Nasty stuff

- It is possible to detect the presence of ptrace (used by gdb). See "A Little Journey to the Wonderful World of ptrace." ([ptr]).
- Jumps to the middle of instructions. Ex: MOV eax,0xd4ff1234 has OP code b8 34 12 ff d4. The OP code of CALL esp is ff d4 which we find three bytes in.
- There's nothing natural about the way gcc (and other compilers) builds an ELF file. It is possible to produce a statically linked and stripped executable. That means (almost) no symbols and sections, just a big chunk of code.

Shellcode

A shellcode is a small piece of machine code. It is typically designed to give the attacker a shell, which she can use to access the host.

A shellcode is *not* an executable. You operating system will not be able to run it as-is. Use the program “demo” (download from the website) to test your shellcode.

A typical shellcode is

- Tiny - often less than 100 bytes.
- NUL byte free (in a C string a NUL byte means the end of the string).
- Position independent (most of the time you don't know where your shellcode ends up, and when you do its program specific).
- Relies as little as possible on its environment (memory layout, memory position, etc.).

Sometimes a host will reject certain input. For example non-printable characters. It is possible to write shellcode using only printable characters. The Internet is filled with coders/decoders.

Kinds of shellcode

See Wikipedia ([Wikipedia b]). And for some really leet stuff see “English Shellcode” ([Mason 09]).

- Local (just run execve(/bin/sh)).
- Remote.
    - Bindshell (attacker connects).
    - Connect-back (shellcode connects).
- Staged. A small shellcode locates (and/or assembles) the real shellcode. An egg-hunter searches memory for the shellcode. Sometimes the shellcode's position can be calculated from pointers in known locations.
- Downloader.

Architecture

As mentioned we focus on Intel X86 32 bits processors. Officially it's called IA32 ([Wikipedia a]), but is also referred to as X86 or i386.

The AMD64 architecture is referred to as X86-64, IA32e, EM64T or X64. It is *not* the same as IA64 (used in Intel Itanium).

Position independent shellcode

Often we don't know where the shellcode is located.

We have two options:

- Find the absolute address.
- Not rely on knowing the address.

Finding the address

This would be great:

```
bedab055: mov eax,eip
...
bedab100: add eax,0x1ab
...
bedab200: [DATA]
```

But there is not such an instruction in X86.

Trampoline

```
da7aba5e: eb 5e              jmp 0x5e
...
da7aba71: 58                 pop eax
...
da7ababe: e8 9d ff ff ff     call -0x63
```

The NOP sled

If we don't know the exact address of the shellcode but have an idea of where it is we can use a NOP sled.

```
90   NOP
90   NOP
90   NOP
90   NOP
...
90   NOP
90   NOP
     [shellcode]
```

Pushing a string on the stack

If we need a string and we don't know where the shellcode is loaded, we can store the string in a known location. The stack is a good option.

```
bedabb1e: 68 73 73 77 64    push "sswd"
bedabb23: 68 63 2f 70 61    push "c/pa"
bedabb28: 68 0a 2f 65 74    push (0xa | ("/et" << 8))
bedabb2d: 68 6f 72 6c 64    push "orld"
bedabb32: 68 6f 2c 20 57    push "o, W"
bedabb37: 68 48 65 6c 6c    push "Hell"
bedabb3c: 89 e0             mov eax,esp
```

Register EAX points to the string "Hello, World\n/etc/passwd".

Controlling registers

Setting EAX to zero

```
5a11b0a7: b8 00 00 00 00    mov eax,0x0

600dbea7: 31 c0             xor eax,eax

7bea71e5: 29 c0             sub eax,eax

da7aba5e: b8 ff ff ff ff    mov eax,-0x1
da7aba63: 40                inc eax

badbaded: 89 d8             mov eax,ebx
badbadef: 31 d8             xor eax,ebx
```

Setting EDX to zero if EAX is non-negative

```
601dc0de: 99                cdq
```

Setting EAX, ECX and EDX to zero

```
be5770ad: 31 c9             xor ecx,ecx
be5770af: f7 e9             imul
(ECX := 0, EDX:EAX := EAX * ECX)
```

Setting EAX to an arbitrary value

```
17570a57: b8 33 22 11 00    mov eax,0x00112233

fa57c0de: b8 32 23 10 01    mov eax,0x01102332
fa57c0e3: 35 01 01 01 01    xor eax,0x01010101

c01dca75: b8 03 32 21 10    mov eax,0x01122330
c01dca7a: c1 c8 04          ror eax,0x4

deadca75: b8 cc dd ee ff    mov eax,0xffeeddcc
deadca7a: f7 d0             not eax
```

Setting a register to a value less than 128

```
1116e718: 6a 17             push byte 0x17
1116e718: 58                pop eax
```

Zero if even

```
        : c1 e0 1f          shl eax,31
```

Zero if positive

```
        : c1 e8 1f          shr eax,31

        : c1 f8 1f          sar eax,31
```

Stack buffer overflow exploits

Read "Smashing the Stack for Fun and Profit" ([One 96]).

Local variables are allocated on the stack. In the typical example a buffer is overflowed in order to overwrite the return address.

Inside a function

```
int myfun (int a, int b, inc) {
  int x, y;
  ...
```

```
cc cc cc cc    [ebp - 0x10] : third argument
bb bb bb bb    [ebp - 0xc]  : second argument
aa aa aa aa    [ebp - 0x8]  : first argument
dd cc bb aa                 : return address (0xaabbccdd)
88 77 ff bf                 : saved base pointer (0xbfff7788)
xx xx xx xx    [ebp + 0x4]  : first local variable
yy yy yy yy    [ebp + 0x8]  : second local variable
```

Look out for strcpy and friends, and loops controlled by input.

Sweet!

```
gdb -args ./stack ‘perl -e ’print "A"x1030’‘
(gdb) run
```

(A good sign that you're controlling EIP)

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ??  ()
```

Now what?

So we control EIP, now what?

If the stack is not randomised we can use a NOP sled and return directly.

Nowadays – unfortunately – the stack randomised. We don't know where it's located.

Return to library

If we now the address of a library we can return to it, (mis)using OP codes. Jump and call instructions can be useful. Especially JMP esp and CALL esp.

If you can't find the code you need, you can sometimes build it from several places in the library. Example (EDX holds the address of the shellcode):

```
08048526:   89 d0   mov eax,edx
08048528:   5e      pop esi
08048529:   c3      ret
...
080486d7:   ff d0   call eax
```

```
...
d7 86 04 08
41 41 41 41
26 85 04 08  < ESP
```

Jump to EAX

```
ff e0               jmp eax


ff d0               call eax


50                  push eax
c3                  ret


89 c3               mov ebx,eax
...
ff d3               call ebx


60                  pusha
...
81 ec 1c 00 00 00   sub esp,0x1c
...
c3                  ret
```

Be creative!

OP codes are everywhere, and some codes have multiple meanings depending on alignment. Sometimes you will find useful OP codes in text string or icons. Look around.

Stack cookies

A stack cookie (or a stack canary as it is often called) is a value pushed onto the stack at the beginning of a function call. After the function has executed the value is inspected. If it has changed the program aborts. GCC and Microsoft Visual Studio implements stack cookies since 1997.

Heap buffer overflow exploits (using dlmalloc)

See "A Memory Allocator" ([Lea 96]).
You will find the source code for Doug Lea's memory allocator in malloc.c in todays exercises.

Memory allocators must find a balance between speed and resilience to errors. Speedy allocators are good for us.

A memory allocator must have some internal data structure to keep track of allocated memory. If we can tamper with this data structure we might be able to get the allocator to help us.

I will not give a recipe for exploiting dlmalloc. Look at the source code, it's nicely documented.

Writing exploits

Write self-documenting maintainable code.

```
#include <stdint.h>
...

unsigned char shellcode[] = {0x6A, 0x0E, 0x5A, ...};
struct {
  unsigned char buf[64];
  uint32_t ebp;
  uint32_t ret0;
  uint32_t esi;
  uint32_t ret1;
  char nul;
} __attribute__((packed)) overflow;

memset(&overflow, 'A', sizeof(overflow));
memcpy(overflow.buf, shellcode, sizeof(shellcode));
overflow.ret0 = 0x08048526;
overflow.ret1 = 0x080486D7;
overflow.nul = '\0';
...
```

Other stuff

See [Erickson 08, Koziol 04].

- Format string exploits.
- Returning into libc (and libc chaining).

Commands

objdump -d Disassemble.

objdump -D Disassemble all.

objdump -h List section header.

nasm -f elf Assemble ELF object file.

gcc -m32 -fno-stack-protector Compile to X86 32 bits without stack protection.

grep "‘perl -e ’print \"\xff\xd4\"’‘" [file] Look for CALL esp in file.

...

Proactive Computer Security

Block 4, 2011.

Look it up on Absalon. Lots of useful slides and links.

Bibliography I

Jon Erickson.
Hacking: The art of exploitation.
No Starch Press, 2nd edition, 2008.

Intel.
Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture.
1999.
Available from: http://developer.intel.com/design/pentiumii/manuals/243190.htm.

Intel.
Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual.
1999.
Available from: http://developer.intel.com/design/pentiumii/manuals/243191.htm.

Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta & Riley Hassell.
The shellcoder's handbook: Discovering and exploiting security holes.
John Wiley & Sons, 2004.

Doug Lea.
A Memory Allocator.
1996.
Available from: http://g.oswego.edu/dl/html/malloc.html.

Bibliography II

Joshua Mason, Sam Small, Fabian Monrose & Greg MacManus.
English shellcode.
In Ehab Al-Shaer, Somesh Jha & Angelos D. Keromytis, editeurs, ACM Conference on Computer and Communications Security, pages 524–533. ACM, 2009.
Available from: http://www.cs.jhu.edu/~sam/ccs243-mason.pdf.

Aleph One.
Smashing the Stack for Fun and Profit.
1996.
Available from: http://insecure.org/stf/smashstack.html.

A Little Journey to the Wonderful World of ptrace.
Available from: http://eigenco.de/releases/ptrace.txt.

William Swanson.
The Art of Picking Intel Registers.
2003.
Available from: http://www.swansontec.com/sregisters.html.

Wikipedia.
IA-32.
Available from: http://en.wikipedia.org/wiki/IA-32.

Wikipedia.
Shellcode.
Available from: http://en.wikipedia.org/wiki/Shellcode.