Software Security

Aka
'Hands-On Hacking'

Nick Nikiforakis
nick.nikiforakis@cs.kuleuven.be

Athens Programme
KU Leuven 2013
Part I

Background
Compiled languages
  - Why do we need them?
  - Why are they here to stay?
What are common problems?
C language
x86 Assembly
Memory segmentation
GDB
When do I get to hack?

- Before you can hack something you first need to understand it
- Hacking without understanding
  - != 1337
NOT part of our talk
Compiled Languages

- The programmer’s source code is translated to machine code by the compiler
- Machine code runs directly on top of hardware
- No run-time interpreters
  - (java, python, perl)
Compiled Languages:

Advantages

- Speed
- Control

Disadvantages

- Longer build cycles
- Little to no enforced security
  - No interpreter to enforce safe operations
- Potential to create havoc
The Need of Compiled Languages

- Native speed will always be needed
  - Operating System kernels
  - Time-dependent tasks
- Common applications in Windows & Linux mostly written in compiled languages
  - Adobe Reader, Firefox, Chrome, Internet Explorer...
Contents

- Compiled languages
  - Why do we need them?
  - Why are they here to stay?
- What are common security issues?
- C language
- x86 Assembly
- Memory segmentation
- GDB
Common Security Issues

- Unsafe pointer operations
  - Reading from places where you shouldn’t
  - Writing to places where you shouldn’t
  - Calling things you shouldn't be calling

More in Part 2
C Language

- Linux Kernel written in C
  - With some parts in assembly
- Second most-common language
  - C and C++, 25% of all programs written
- Static type checking
  - NOT memory safe
Compiled languages
- Why do we need them?
- Why are they here to stay?
What are common problems?
C language
x86 Assembly
GDB
Memory segmentation
Assignment
X86 processor

- Most common processor-type in desktop/laptop/server environments
- X86 Instruction set
  - The CPU's language
  - Operation <destination>, <source>
- Native programs → x86 instructions
- 32bit, 64bit
X86: Registers

- X86 has several registers
  - Internal fast memory
  - Can be viewed as internal variables
X86: General Purpose Registers

- EAX: Accumulator
- ECX: Counter
- EDX: Data
- EBX: Base
- ESP: Stack Pointer
- EBP: Base Pointer
- ESI: Source Index
- EDI: Destination Index
Special Registers

- EIP: Instruction Pointer
- EFLAGS: Status Register
X86: Common Instructions

- Moving data from one 'place' to the other
  - `mov eax, 0x1234`
  - `mov ebx, ecx`
  - `mov DWORD PTR [esp+0x1c], 0x0`

- Arithmetics
  - `add esp, 0x1`
  - `sub ebp, 0x2`
  - `inc eax`
  - ...

Common Instructions

- Comparisons
  - cmp eax, ebx
  - cmp ecx, 0x45

- Branching
  - Jumps: jle, jeq, jne, jmp
    - jmp 0x80abc44
  - Function calls: call

- Misc:
  - lea eax, [ebp - 4]
Contents

- Compiled languages
  - Why do we need them?
  - Why are they here to stay?
- What are common problems?
- C language
- x86 Assembly
- GDB
- Memory segmentation
- Assignment
GDB

- GDB: Gnu Project Debugger
- Start/Stop/Pause an executing program
- Investigate/Alter
  - Registers
  - Memory locations
GDB Common commands

- **break**: Sets a breakpoint to pause the execution when a specific statement it is reached
- **x**: Examines the contents of memory locations/registers
- **nexti**: execute next instruction after a breakpoint
- **continue**: continue execution after a breakpoint
Memory Segmentation

- Each process when loaded to memory has a number of different segments:
  - **Text**: Code of program (r)
  - **Data**: Global Initialized & Static variables (r,w)
  - **BSS**: Global Un-initialized variables (r,w)
  - **Heap**: Pool for dynamically allocating memory
  - **Stack**: Used by each called function for control and non-control data
Process view

2^{32} - 1

OS Kernel

Stack

Heap

BSS

Data

Text

High addresses

Low addresses

0
Stack

- Used by each called function for control and non-control data
- An orchestrated dance using ebp, esp and plenty of memory
22: ...
23: foo(a,b,c);
24: a = a + 3;
25: ...

```c
void foo(int a, int b, int c) {
    int i = 0;
    int array[10];
    ....
    return;
}
```
Putting it all together
Part II

Stack-based Buffer overflows
Buffer: A region of memory which can hold data
- char c_array[100];
- int i_array[20];
- int *ip_array[10];

Overflow: Add more things than something can contain
Buffer overflow

- What happens when you try to copy into a buffer, more data than it can handle?
  - Spills out to adjacent memory

- `void foo(char *input){
    int i =0;
    char array[10];

    strcpy(array,input);
}
`
void foo(char *input){
    int i = 0;
    char array[16];
    strcpy(array, input);
}

foo('AAAAAAAAAA');
foo('AAAAAAAAAAAAAAAAAA');
foo('AAAAAAAAAAAAAAAAAAAAAAAAAA');
foo("\x08\x4A\x6B........\x08\xff\xbf");
Standard Way

Shellcode

Return Addresses

Buffer1

Int  SFP  RA
Exploitation of a vulnerable C program
Contents

- Overflows in different segments
- Format-string vulnerabilities
- Writing your own shellcode
An overflow doesn't have to occur on the stack

Heap Overflow

```
function la(char array[]){
    int *ptr1 = malloc(24);
    int *ptr2 = malloc(48);

    strcpy(ptr1,array);
    return;
}
```
Heap Overflows

- function la(char array[]){
  int *p1 = malloc(24);
  int *p2 = malloc(48);
  strcpy(ptr1, array);
  return;
}
Overflows everywhere

- Wherever a buffer can be placed, a buffer overflow is possible
  - .data
  - .bss
  - Stack
  - Heap
  - Kernel
Format string attacks

- There is no overflowing here
- Misusing popular functions
  - printf
  - fprintf
  - snprintf
  - ....
- printf("i = %d, name = %s\n",i,name);
Format string attacks

- `printf("i = %d, name = %s\n", i, name);`

What happens if you forget a parameter?
So far we have been using ready-made shellcode

```c
char shellcode[] = ":0eb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\x89\xc7\xff\xff\xff/bin/sh";
```

But what is it exactly?
Shellcode

- Shellcode is the hex representation of assembly instructions that will allow us to spawn a shell
- However, shellcode is not limited to spawning a shell
  - Everything you can do with a regular program, you can do with shellcode
    - Write to files, delete files, spawn processes etc.
Shellcode

- However, you can't write shellcode the same way you would write assembly.

- Shellcode needs to be PIC:
  - You can only use relative addresses.
  - You can't make any assumptions of where your code will be placed.
  - Your code shouldn't contain NULL bytes.
    - When needed you have to create them on the fly.
Shellcode

- Use strace to learn about system calls
- "man" the system-call you want to learn about how to call it
- Write code that will not contain NULL bytes

DEMO
Part III

Deployed Countermeasures & Ways around them
Countermeasures

- Software was and still is plagued by bugs that can be used as vulnerabilities
- Built-in countermeasures in your OS
  - StackGuard, SSP
  - ASLR
  - Non-exec stack & heap
StackGuard

- Remember the old stack?

```c
void foo(input){
    Int i;
    Char array[16];
    strcpy(array,input);
}
```
StackGuard modification

- StackGuard inserts a canary value...

```c
foo(input);

void foo(input){
  Int i;
  Char array[16];
  strcpy(array,input);
}
```

![Image of stack frames and canary value](image.png)
StackGuard

- Good, but not great
  - Variables between the buffer and the canary can still be overflowed without errors
  - Saved frame pointer can be overridden so that when it is popped, it will pop in a fake stack frame that will call attacker code...

- Solution?
  - ProPolice aka Stack Smashing Protection (SSP)
SSP modification

- Canary value and rearrangement of variables

```c
void foo(input){
    Int i;
    Char array[16];
    strcpy(array,input);
}
```

**DEMO**
ASLR

- Address Space Layout Randomization
- Depending on the OS and Kernel one or more of the following can be randomized:
  - Base of stack
  - Base of Heap
  - Base of shared libraries
- If the binary is compiled as PIE (Position Independent Executable), even the .text can be randomized
Non-executable stack & heap

- **Recipe for traditional buffer overflows:**
  - Place shellcode in buffer
  - Overwrite return address/function pointer/etc. with address of buffer
  - **CPU will execute your code**

- **W^X**
  - A memory page can be writable or executable but not both
Ways to go around them

- **SSP:**
  - Attack the heap/data segments where no canaries are placed
  - Find out the canary and rewrite it with its identical value when doing the overflow
    - Brute-force
    - Buffer-overread
    - Format string vulnerabilities
  - If applicable, attack another character buffer
  - If applicable, attack a struct containing a character buffer
Ways to go around them

- **ASLR**
  - De-randomize the stack/heap using a buffer over-read/format-string vulnerability
  - Use relative jumping instead of absolute
  - Heap-Spraying, many instances of shellcode
  - Depending on what is randomized, use existing instructions in non-randomized segments instead of your own shellcode (Return-into-libc, Return-Oriented Programming)
Ways to go around them

- W^X
  - Use existing instructions placed in executable memory (Return-into-libc and ROP)
  - If a Just-in-Time (JIT) compiler is present, use it to place your code in executable memory pages
    - Environments with JIT compilers: JavaScript, Flash, etc.
Every program that you compile is automatically linked to libc

- Implementation of all the ready-made functions you use (e.g. printf, scanf and fgets)
- Many ready-made functions that are linked to your executable are of much interest to an attacker
  - System, exec*
foo(input);

void foo(input){
  Int i;
  Char array[16];
  strcpy(array,input);
}

Return-into-libc

... 

input

Return_Addr_Main

Saved Frame Ptr

i

array[15]

array[14]

array[13]

...

array[0]
void foo(input){
    Int i;
    Char array[16];
    strcpy(array,input);
}

Return-into-libc
Return Oriented Programming

- Generalization of Return-into-libc
- Using the stack to call bits and pieces of assembly that will, in total make something usefull
- ESP is now essential used as the EIP
Return-Oriented Programming is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from text segments.

From: Practical Return-Oriented Programming, Dino A. Dai Zovi
Return Oriented Programming

EAX = SMTH
EBX = SMTH
ECX = SMTH

0x80abdea0
0x309
0x80345677
&"/tmp/lala"
0x80abddaa
8
0x80abcdee

0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
Return Oriented Programming

EAX = SMTH
EBX = SMTH
ECX = SMTH

... 0x80abdea0
... 0x309
... 0x80345677
... &”/tmp/lala”
... 0x80abddaa
... 8
... 0x80abcdee

High

ESP

Low

EIP

0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
## Return Oriented Programming

### Example Code Snippet

```
EAX = 8  
EBX = SMTH  
ECX = SMTH  

0x80abdea0  
0x309  
0x80345677  
&"/tmp/lala"  
0x80abddaa  
8  
0x80abcdee  

0x80abdea0: int 0x80;  
```

### Memory Layout

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80abdea0</td>
<td>0x80abdea0</td>
</tr>
<tr>
<td>0x309</td>
<td>0x309</td>
</tr>
<tr>
<td>0x80345677</td>
<td>0x80345677</td>
</tr>
<tr>
<td>&amp;&quot;/tmp/lala&quot;</td>
<td>/tmp/lala</td>
</tr>
<tr>
<td>0x80abddaa</td>
<td>0x80abddaa</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0x80abcdee</td>
<td>0x80abcdee</td>
</tr>
</tbody>
</table>
```

### Instructions

```
0x80abdea0: int 0x80;  
0x80abddaa: pop $ebx;  
0x80abddab: ret;  
0x08abcdee: pop $eax;  
0x08abcdef: ret;  
0x80345677: pop $ecx;  
0x80345678: ret;  
```

### Registers

- **EAX**: 8
- **EBX**: SMTH
- **ECX**: SMTH
### Return Oriented Programming

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80abdea0</td>
<td></td>
</tr>
<tr>
<td>0x80345677</td>
<td></td>
</tr>
<tr>
<td>&amp;&quot;/tmp/lala&quot;</td>
<td></td>
</tr>
<tr>
<td>0x80abddaa</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0x80abcdee</td>
<td></td>
</tr>
</tbody>
</table>

- **EAX = 8**
- **EBX = SMTH**
- **ECX = SMTH**

---

```
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...```

---

- **ESP**
- **EIP**

---

High

Low
Return Oriented Programming

EAX = 8
EBX = "/tmp...
ECX = SMTH
Return Oriented Programming

EAX = 8
EBX = &"/tmp...
ECX = SMTH

```
0x80abdea0
0x309
0x80345677
&"/tmp/lala"
0x80abddaa
8
0x80abcdee
```

```
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
Return Oriented Programming

<table>
<thead>
<tr>
<th>ESP</th>
<th>0x80abdea0</th>
<th>0x80345677</th>
<th>&amp;&quot;/tmp/lala&quot;</th>
<th>0x80abddaa</th>
<th>8</th>
<th>0x80abcdee</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x309</td>
<td>0x80345677</td>
<td>&amp;&quot;/tmp/lala&quot;</td>
<td>0x80abddaa</td>
<td>8</td>
<td>0x80abcdee</td>
<td></td>
</tr>
</tbody>
</table>

EAX = 8
EBX = &"/tmp..."
ECX = 0x309

0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef: ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
Return Oriented Programming

```
EAX = 8
EBX = "/tmp...
ECX = 0x309

0x80abdea0:
0x80345677: pop $ecx;
0x80345678: ret;
...
0x08abcdee: pop $eax;
0x08abcdef : ret;
...
0x80abddaa: pop $ebx;
0x80abddab: ret;
...
0x80abdea0: int 0x80;
...
```

Diagram:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>ESP</th>
<th>EIP</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80abdea0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x309</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x80345677</td>
<td></td>
<td></td>
<td>8</td>
<td>&amp;&quot;/tmp/lala&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x80abddaa</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x80abcdee</td>
<td></td>
</tr>
</tbody>
</table>
```

ESP points to 0x80abdea0, which contains instructions to pop EAX and EBX, and then ret to 0x80345678.

EAX = 8
EBX = "/tmp...
ECX = 0x309
Conclusion

- Now you know it all
  - Just kidding... you know the very basics, but it's a really good start

- Compiled programs, are not going away any time soon
  - Thus the attacks against them are not going away any time soon
  - Attacks are of increasing difficulty and rarity, but still very much possible
If time allows...
Academic Countermeasures
HeapSentry
Kernel-assisted Protection against Heap Overflows

Nick Nikiforakis, Frank Piessens, Wouter Joosen
DIMVA 2013
Sentry?

- **Sentry**: A soldier stationed to keep guard or to control access to a place
Roadmap

• Motivation
• Design of HeapSentry
• Evaluation
• Conclusion
Problem

• Overflows are still with us
  – 418 reported overflows for 2012
  – 135 were heap-based (32.3%)
  – 3rd in SANS Top 25 Most Dangerous Programming Errors
  – Most high-profile attacks usually involve one
Modern OSes

- Orthogonal runtime & compile-time countermeasures:
  - ASLR
  - W^X
  - Protection of stack frames
  - StackGuard, ProPolice
An old friend…

- The “Stack” part of a process used to be the most attacked one
  - Classic stack-smashing
  - Overwriting local args & function parameters

- Now its probably the least one
  - Canary in the way
  - Copied parameters and local args above the buffers
A “new” friend

• What about the heap?
  – Segment where all dynamic memory lives

• Who protects it?
  – No one....
  – most of the time

• An attacker can overflow freely from one heap object to the next
  – If he is detected, he will be detected at the next free ... on that block
Roadmap

• Motivation
• Design of HeapSentry
• Evaluation
• Conclusion
HeapSentry characteristics

- Cooperative approach between a memory allocator and the OS’ kernel to detect heap overflows
- Independent of underlying OS and memory allocator
Attacker model and requirements

• Independent of attacked object
  – Meta-data, function pointers, non-control data...

• Independent of attacking methodology
  – Injection of code, ret2libc, ROP-programming

• No need for source code
Core Idea

• Add a unique random canary at the end of each allocated block
• Register this location and the original value to the kernel
• Have the kernel check the liveness of the canaries
  – Kill the process if a canary is “dead”
Visually...

Heap

[Diagram of heap with several boxes and ellipsis]
Visually...

Heap

KERNEL SPACE

Canary

Original Location

Value

USER SPACE

<table>
<thead>
<tr>
<th>C1</th>
<th>V1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>V2</td>
</tr>
<tr>
<td>C3</td>
<td>V3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Cn</td>
<td>Vn</td>
</tr>
</tbody>
</table>
HeapSentry-U

- Intercept all calls to dynamic memory functions
  - malloc, calloc, realloc, free
- Generate random canaries and communicate to kernel
HeapSentry-U: malloc

• `malloc(42)`
  – `ptr = realloc(42 + sizeof(int));`
  – `*(ptr + 42) = new random canary`
  – `syscall(ptr + 42) //HeapSentry-K`
  – return to caller
HeapSentry-U: free

- free(ptr)
  - canary_loc = ptr + sizeof(block) - sizeof(canary);
  - syscall(canary_loc);
  - real_free(ptr);
  - return to caller
• Kernel module
  – Hijack the execution flow just before the dispatch of each individual system call
  – Exchange messages with HeapSentry-U
  – Check the liveness of canaries before the execution of a program-requested system call
Where?
Need for speed

• In its basic design, HeapSentry will execute one system call every time a dynamic memory function is called
  – Too much trapping to the kernel
  – Too many canaries to check at each system call

• Optimizations:
  1. System call categorization
  2. Grouping of operations
• **The goal**: Check less canaries at each system call invocation

• Are all system calls dangerous?
  – What is the chance that a system call is requested by an attacker

• Classes:
  1. High Risk – scan all
  2. Medium Risk – scan some
  3. No Risk – don’t scan
<table>
<thead>
<tr>
<th>Category</th>
<th>System call</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Risk</td>
<td>fork</td>
</tr>
<tr>
<td></td>
<td>execve</td>
</tr>
<tr>
<td></td>
<td>chmod</td>
</tr>
<tr>
<td></td>
<td>open</td>
</tr>
<tr>
<td>Medium Risk</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>write</td>
</tr>
<tr>
<td></td>
<td>mount</td>
</tr>
<tr>
<td></td>
<td>mmap</td>
</tr>
<tr>
<td>No Risk</td>
<td>brk</td>
</tr>
<tr>
<td></td>
<td>getpid</td>
</tr>
<tr>
<td></td>
<td>chdir</td>
</tr>
<tr>
<td></td>
<td>close</td>
</tr>
</tbody>
</table>
Grouping operations

- **The goal**: trap less times to the kernel
- Report to the kernel in batches instead of every time
  - e.g. here are 50 canaries and their values
  - e.g. check these 50 canaries so that I can free their blocks

- Result: 50 times less system calls over the basic HeapSentry design
Safety issues?

- Consider a memory-intensive program with an average of 100,000 active heap objects.
- At any point in time:
  - A minimum of 99.95% of canaries are stored in the kernel.
  - A maximum of 0.05% of canaries are still unreported.
  - Attacker must locate buffers and remove before the process goes to the kernel.
    - Typical in multithreaded programs.
  - Guard pages around the buffers.
Control flow
• Motivation
• Design of HeapSentry
• Evaluation
• Conclusion
Security Evaluation

- Analyze 100 shellcode samples from shell-code.org
  - Removed 5 with non-critical payload
  - launching shells, killing IDSs, reading & editing system files
- All 95 where using at least one high-risk system call
  - HeapSentry would stop all of them in case of an attack
Performance

- SPEC CPU2006

AVG: 11.6% added overhead
Limitations

• As with all canary-based approaches, if the canary is not overwritten, the heap overflow will not be detected
  – Overflow within the same allocated object
  – Direct overwrite
Roadmap

- Motivation
- Design of HeapSentry
- Evaluation
- Conclusion
Conclusion

• Buffer overflows are still an important problem
• The heap has received less attention than other parts of a process’ address space
• HeapSentry, a cooperative approach between the memory allocation library and the kernel based on canaries