The Morris Internet Worm source code

This disk contains the complete source code of the Morris Internet worm program. This tiny, 99-line program brought large pieces of the Internet to a standstill on November 2nd, 1988.

The worm was the first of many intrusive programs that use the Internet to spread.
Long ago – late 1980s

- On November 2, 1988 the Morris Worm was released
  - Mainstream media attention
  - Conviction under the Computer Fraud and Abuse Act
  - First well-known program exploiting a buffer overflow

http://en.wikipedia.org/wiki/Morris_worm
25 years later

Memory errors and memory corruption vulnerabilities are still an issue!
This talk is about

- Why these bugs are still a concern
- How exploits work
- Modern defenses
Motivation
Today, 2014

- Memory errors are still a problem
  - “Unsafe” languages like C/C++ very popular
    - Prediction: C/C++ will be with us for a long time
    - Yes, there are alternatives... sometimes
  - Criminals found ways of monetization
  - Software systems are gaining complexity

http://www.langpop.com/
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
**Terminology**

**Exploit**

“An exploit is a piece of software, a chunk of data, or a sequence of commands that takes advantage of a bug, glitch or vulnerability in order to...”

**Zero-Day Attack**

“A zero-day attack or threat is an attack that exploits a previously unknown vulnerability in a computer application, ...”

http://en.wikipedia.org/wiki/Exploit_(computer_security)
http://en.wikipedia.org/wiki/Zero-day_attack
Attacks

Victim

- Runs a vulnerable web browser or PDF reader
- Sends a malicious PDF attachment by email
- GET /index.html HTTP/1.1
  Host: www.vulnsite.com
  Keep-Alive: 300
  Connection: keep-alive
  Cookie: CID=r2t5uvjq43
  5r4q7ib3vtdjgq120f83jf8...
  ... <binary data>

Attacker

- Runs a malicious web server serving HTML documents that trigger a memory error within the web browser
- Exploits memory error within vulnerable victim software
- $>./exploit 192.168.1.28
- Runs an exploit that triggers a memory error within the web server software
Arbitrary Code Execution
Modern software stack

- **Application**
- **Web Browser**
- **Client-side Script**
- **Java VM**
- **Java Application & Libraries**
- **Libraries & Tools**
- **Operating System**
- **Hardware**
Modern software stack

Potentiall prone to memory errors & corruption
The Internet of “Memory Unsafe” Things
Common Vulnerabilities and Exposures

% of total CVEs

- Memory Errors: 36.29 %
- XSS + CSRF: 0.1 %
- SQL Injection: 19.84 %
- Memory Errors: 8.05 %
- XSS + CSRF: 2.75 %
- SQL Injection: 3.34 %
- Memory Errors: 31.39 %
- XSS + CSRF: 0 %

Memory Errors: CWE-119, CWE-399 "use after free", CWE-189 in High / XSS + CSRF: CWE-79, CWE-352 / SQL Injection: CWE-89

https://cve.mitre.org/
Finding vulnerabilities

- Finding exploitable errors is not trivial
  - Static and dynamic analysis, testing, reviews

Edsger W. Dijkstra, 1969:

“Testing shows the presence, not the absence of bugs.”
Thinking about a career change?

The 2014 targets are:

**Browsers:**
- Google Chrome on Windows 8.1 x64: $100,000
- Microsoft Internet Explorer 11 on Windows 8.1 x64: $100,000
- Mozilla Firefox on Windows 8.1 x64: $50,000
- Apple Safari on OS X Mavericks: $65,000

**Plug-ins:**
- Adobe Reader running in Internet Explorer 11 on Windows 8.1 x64: $75,000
- Adobe Flash running in Internet Explorer 11 on Windows 8.1 x64: $75,000
- Oracle Java running in Internet Explorer 11 on Windows 8.1 x64 (requires click-through bypass): $30,000

**“Exploit Unicorn” Grand Prize:**
- SYSTEM-level code execution on Windows 8.1 x64 on Internet Explorer 11 x64 with EMET (Enhanced Mitigation Experience Toolkit) bypass: $150,000*

---

http://www.forbes.com/sites/andygreenberg/2012/03/23/shopping-for-zero-days-an-price-list-for-hackers-secret-software-exploits/
Thinking about a career change?

---

**ADOBE READER** $5,000-$30,000  
**MAC OSX** $20,000-$50,000  
**ANDROID** $30,000-$60,000  
**FLASH OR JAVA BROWSER PLUG-INS** $40,000-$100,000  
**MICROSOFT WORD** $50,000-$100,000  
**WINDOWS** $60,000-$120,000  
**FIREFOX OR SAFARI** $60,000-$150,000  
**CHROME OR INTERNET EXPLORER** $80,000-$200,000  
**IOS** $100,000-$250,000

---

The 2014 targets are:

**Browsers:**
- Google Chrome on Windows 8.1 x64: $100,000  
- Microsoft Internet Explorer 11 on Windows 8.1 x64: $100,000  
- Mozilla Firefox on Windows 8.1 x64: $50,000  
- Apple Safari on OS X Mavericks: $65,000

**Plug-ins:**
- Adobe Reader running in Internet Explorer 11 on Windows 8.1 x64: $75,000  
- Adobe Flash running in Internet Explorer 11 on Windows 8.1 x64: $75,000  
- Oracle Java running in Internet Explorer 11 on Windows 8.1 x64 (requires click-through bypass): $30,000

“Exploit Unicorn” Grand Prize:
- SYSTEM-level code execution on Windows 8.1 x64 on Internet Explorer 11 x64 with EMET (Enhanced Mitigation Experience Toolkit) bypass: $150,000*

---

Publicly known

Individuals, groups or security researchers

Public directories & databases

Bounty & reward programs

Competitions
Publicly known

Individuals, groups or security researchers

Public directories & databases

Bounty & reward programs

Competitions
In the end

We have to accept the residual risk...

...but to manage the risks we have to understand the attack techniques and the effectiveness of available defenses!
Memory Errors & Vulnerabilities
Memory errors & vulnerabilities

- Come in various forms with different root causes
- Allow attackers to corrupt memory in a more or less controllable way
  - Worst case: attackers gain arbitrary code execution
- Exist in programs written in “unsafe” languages that do not enforce memory safety
“Unsafe“ languages

- Allow low-level access to memory
  - Typed pointers & pointer arithmetic
  - No automatic bounds checking / index checking

- Weakly enforced typing
  - Cast (almost) anything to pointers

- Explicit memory management
  - Like malloc() & free() in C
“Unsafe” languages - C

```c
#include <stdio.h>
#include <math.h>

long computation(int x, int y, int z, double f) {
    return (long)(((x*y)/z)*sin(f));
}

void main()
{
    *((int*)computation(32, 64, 2, M_PI/6)) = 128;
    return;
}
```

shell:~$ gcc -o segfault segfault.c -lm
shell:~$ ./segfault
Segmentation fault (core dumped)
shell:~$
"Unsafe" languages - C

```c
#include <stdio.h>
#include <string.h>
#define STDIN 0

void vulnFunc() {
    char buf[1024];
    read(STDIN, buf, 2048);
}

void main() {
    printf("read> ");
    vulnFunc();
    return;
}
```

```bash
shell:~$ gcc -o vuln vuln.c -fno-stack-protector
shell:~$ ./vuln
read> hi there!
shell:~$ ./vuln
read>
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
Segmentation fault (core dumped)
shell:~$
```
Types of memory errors

Spatial error

- De-reference pointer that is out of bounds
- Read or write operation

Temporal error

- De-reference pointer to freed memory
- Read operation
Exploiting memory errors

**Spatial error**

- **Attacker supplied data** overwrites/reads data/pointers
- **Array / Object**
- Overwrite data or pointers
- Used or de-referenced later

**Temporal error**

- **Attacker supplied data used as wrong type**
- **Make application allocate memory in the freed area**
- **Used as old type**
Attackers use memory errors to

- Overwrite data or pointers
  - That might be used to overwrite data or pointers
  - Function pointers, sensitive data, index values, control-flow sensitive data etc.

- Leak information
  - E.g., corrupt a length field

- Construct attacker primitives
  - Write primitive (write any value to arbitrary address)
  - Read primitive (read from any address)
  - Arbitrary call primitive (call any arbitrary address)
Types of bugs

- Out-of-bounds bugs / Buffer overflows
  - On stack or heap
- Dangling pointer / Use-after-free
- Integer bugs, signedness bugs
- Format string bugs
- Uninitialized memory
- NULL pointer dereference
- etc.
Memory corruption attack

Victim

Stack

Corrupted
0x41414141

Heap

Code

Attacker

<html>
<head>
</head>
<body>
... <html or javascript that triggers the memory corruption vulnerability> ...
</body>
</html>

Write primitive

*0xe8a0f000 = 0x41414141
Attack types

- Code corruption attack
- Control-flow hijack attack
- Data-only attack
- Information leak

Attack model according to: “SoK: Eternal War in Memory” Laszlo Szekeres, Mathias Payer, Tao Wei, Dawn Song
Attack types

- Code corruption attack
- Control-flow hijack attack
- Data-only attack
- Information leak

Attack model according to: “SoK: Eternal War in Memory” Laszlo Szekeres, Mathias Payer, Tao Wei, Dawn Song
Control-flow hijack attacks

• Most powerful attack

• Hijack control-flow
  • To attacker supplied arbitrary machine code
  • To existing code (code-reuse attack)

• Corrupt code pointers
  • Return addresses, function pointers, vtable entries, exception handlers, jmp_bufs
Normal control-flow

- Basic block
- Direct branch
- Indirect branch
Hijacked control-flow

- Basic block
- Direct branch
- Indirect branch
Control-flow hijack attacks

- Most ISAs support indirect branch instructions
  - E.g., x86 "ret", indirect "jmp", indirect "call"

`fptr` is a value in memory at 0xafe08044
- branch *fptr
Control-flow hijack attacks

```plaintext
fptr is a value in memory at 0xafe08044
- branch *fptr
- fptr was corrupted by an attacker
```

**Attacker goal:** hijack control-flow to injected machine code or to “evil functions”
Branch instructions – Intel x86

- Direct or indirect **call**
  - call 804a450 or call *0x24(%ebp)

- Direct or indirect **jmp**
  - jmp 8049e70 or jmp *0x805a874(%eax,4)
  - There are also conditional jumps (direct)

- **ret**, Indirect by design
  - Take value on top of stack and branch to it

- **call** instructions push address of the next instruction onto the stack so the **ret** instruction can use it
Generalized process layout – user space

**Stack**: contains local data of functions/procedures/methods

**Heap**: contains data managed by the heap manager

**Memory Mappings**: mapped files or anonymous memory mappings

**Libraries**: code and data of dynamically linked libraries

---

**ELF / PE files**
Attack surface

Stack

Heap

Memory Mappings

Libraries

Data

Code

Network API

FileSystem API

IPC

Network

Storage

Process

OS

Network

PDF

IPC

011010001
100100100
111010110

0x00000000

0xffffffff
Attack surface

- Stack
- Heap
- Memory Mappings
- Libraries
- Data
- Code

Network API

File System API

IPC

Storage

Process

Network

Memory Mappings:
- Stack
- Heap

Libraries:
- attacker code & data

Data:
- attacker code & data

Code:
- attacker code & data

Network:
- attacker code & data

Storage:
- attacker code & data

Process:
- attacker code & data

Memory Mappings:
- Stack
- Heap

Libraries:
- attacker code & data

Data:
- attacker code & data

Code:
- attacker code & data

Network:
- attacker code & data

Storage:
- attacker code & data

Process:
- attacker code & data

Memory Mappings:
- Stack
- Heap

Libraries:
- attacker code & data

Data:
- attacker code & data

Code:
- attacker code & data

Network:
- attacker code & data

Storage:
- attacker code & data

Process:
- attacker code & data
Control-flow hijack to injected code

Indirect call to func()

Hijacked indirect call
Non-eXecutable data (NX)

- Make **data** regions **non-executable** (by default)

- Changing protection flags or allocating **rwx** memory still possible (on most systems)
  - Required for JITs
Non-eXecutable data (NX)

- Code regions will be non-writable
  - Else code corruption attacks possible

- Also known as
  - Data Execution Prevention (DEP) on Windows
  - NX, Non-eXecutable Memory on Linux
  - W^X, on OpenBSD
  - Implemented by hardware where available (NX-bit)
NX / DEP – implementation issues

Compatibility

• Binary images need to provide separate sections/segments that can be mapped exclusively as **rw-** OR **r-x**
  • Linker support required

• Self-modifying code not allowed
  • Compiler support required
  • If code is generated just-in-time, explicit **rwx** allocation required
Bypassing NX / DEP

- Only use existing code
  **Code-reuse attack**
  - ret2libc, ret2bin, ret2*
  - Return-oriented programming (ROP)
  - Jump/Call-oriented programming

- Use code-reuse technique to change protection flags
  - Allocate or make memory executable
    - mprotect/VirtualProtect
    - mmap/VirtualAlloc
Code-reuse attack

```c
void vulnFunc() {
    char buf[1024];
    read(STDIN, buf, 2048);
}
```

- Stack-based buffer overflow
  - `%eip` and `%ebp` under attacker control
  - Local variables and buffers under attacker control

Stack during `vulnFunc()`

Stack after `read()`
Code-reuse attack

- Before NX/DEP
  - Return to attacker supplied code
    - Shellcode

- Bypass NX/DEP by using existing code
  - Executable or libraries E.g., mprotect()
Code-reuse attack

- Let's call `mprotect()`
  
  \[
  \text{mprotect(address\_shellcode, 4096, 0x1|0x2|0x4)}
  \]

- \(0x1|0x2|0x4 = \text{RWX}\)

- `mprotect()` will make the stack executable
  - And return to our shellcode
Code-reuse attack

- After `mprotect()` our stack is executable again
  - `mprotect()` will return to our shellcode
- Works well on x86 32bit but on x64 or ARM function parameters are passed over registers
  - Fill registers with parameters before invoking `mprotect()`
Return-oriented programming

- Use available code snippets ending with `ret` instruction
- Called gadgets / ROP chain
- E.g., write primitive
RETURN ORIENTED PROGRAMMING
Return-oriented programming

- Very powerful!
  - Turing complete although not required

- Can be initiated over call or jmp as well

- Need to be in control of memory %esp is pointing to
  - Or make %esp point to area under control

- Also possible with jmp or call gadgets
  - Complicated to keep control and dispatch to the next gadget
  - Generalization: Gadget-oriented programming
Return-oriented programming

• Notes on calling convention

  • x86 32bit, easy, arguments passed over stack
    • If stack attacker controlled

  • x86 64bit & ARM, arguments passed in registers
    • More general purpose registers
    • Calling functions more laborious
    • Copy arguments from attacker data to registers
      • ARM has interesting „pop“ into multiple registers feature
Addresses in memory

- To hijack control-flow or to corrupt memory an attacker **needs to know where things are in memory**
  - Addresses of data or pointers to corrupt
  - Addresses of injected shellcode/payload
  - Addresses of gadgets

- Sometimes it's enough to know the rough location but **most of the time** attackers need the **exact location**
  - Corrupting only least significant bytes i.e. an offset might work in some special cases (but not in general)
Addresses in memory

Once upon a time...

- Addresses were more or less predictable
- Executables and libraries were prelinked to certain addresses
- Stack and Heap base addresses were fixed
  - With differences at runtime for specific locations due to the dynamic behaviour of the process
ASLR

Today most Operating Systems implement Address Space Layout Randomization

- Randomize memory layout of processes to make address prediction or guessing hard

- What can be randomized?
  - OS: Stack, heap and memory mapping base addresses
  - OS, compiler, linker: Executables and libraries
    - Position-independent or relocatable code
Randomization of layout

Use source of randomness to randomize within ranges
ASLR – implementation issues

Compatibility

- In general: Usage of fixed addresses not allowed
  - Hardcoded addresses

- Code should be position-independent or relocatable
  - Linux/ELF:
    - PIC & PIE supported, libraries all PIC, executables sometimes still prelinked
  - Windows/PE:
    - No PIC support! But libraries/executables relocatable!

- x86 32bit PIE/PIC slower, no IP relative data addressing
- Relocated PE images not sharable between processes
ASLR – effectiveness

- Enough entropy?
  - Base range size
  - Alignment constraints
  - Address width (64bit is better than 32bit)

- Randomization strategy
  - Per process, system-wide per boot

- Source of randomness
ASLR – all or nothing

• ASLR only effective without exceptions

• One library/executable without ASLR might already compromise security
  • One datastructure without ASLR as well
Bypassing ASLR

- Low entropy
  - Brute-force addresses (multiple attempts required)

- Memory leaks (information disclosure)
  - Leak addresses to derive base addresses
    - E.g., run-time address pointing into a library
    - Construct and enforce a leak by memory corruption

- Application and vulnerability specific attacks
  - Force predictable memory state
    - Heap-spraying
  - Pointer inference
    - Use a side-channel
  - Avoid using exact addresses
    - Only corrupt least significant bytes i.e. offsets
Memory leak

1. Trigger memory leak
2. Parse response for leaked memory and construct exploit
3. Exploit memory error with tailored exploit

Heap

0xffed4460

Stack

Code

Runs web server software with memory leak and exploitable memory corruption vulnerability

GET /index.html HTTP/1.1
Host: www.vulnsite.com
Keep-Alive: 300
Connection: keep-alive
Cookie: CID=r2t5uvjq435r4q7ib3vtdjq120f83jf8...

HTTP/1.1 200 OK
Date: Fri, 12 Mar 2014 23:59:00 GMT
Content-Type: text/html
Content-Length: 1354
Allow: GET,POST,OPTIONS,HEAD
Cache-Control: public,max-age=30

<html>
<head>...
</head>

GET /index.html HTTP/1.1
Host: www.vulnsite.com
Keep-Alive: 300
Connection: keep-alive
Cookie: CID=r2t5uvjq435r4q7ib3vtdjq120f83jf8...

... <binary data>

1

2

3
**Memory leak**

\[
\text{mprotect} = \text{leaked pointer} - \text{static offset}
\]

\[
0x0ebb0880 = 0x0efa4604 - 0x003f3d84
\]
Memory leak – format string bug

```c
#include <stdio.h>
#include <string.h>

#define STDIN 0

void memLeak() {
    char buf[256];
    scanf("%s", buf);
    printf(buf);
}

void main() {
    printf("echo> ");
    memLeak();
    printf("\n");
    return;
}
```

```
shell:~$ gcc -o memleak memleak.c
memleak.c: In function ‘memLeak’: 
memleak.c:9:2: warning: format not a string literal and no format arguments [-Wformat-security]
shell:~$ ./memleak
echo> hi
hi
shell:~$ ./memleak
echo> %llx,%llx,%llx,%llx,%llx,%llx,%llx,%llx
1,7fabf517cac0,a,ffffffff,0,6c6c252c786c6c25,252c786c6c252c78,786c6c252c786c6c
shell:~$
```
Heartbleed – CVE-2014-0160 - OpenSSL

http://en.wikipedia.org/wiki/Heartbleed
Simplified Heartbleed explanation by FenixFeather licensed under CC
DEP & ASLR

DEP & ASLR are generic defenses

- Exploitation becomes harder for all vulnerability classes & attack techniques

- Together quite effective
  - If implemented correctly and used continuously

- Injecting code and corrupting pointers with exact addresses is in general desirable for attackers
DEP & ASLR

• But DEP & ASLR are not enough

• A determined attacker will use code-reuse techniques and memory leaks to bypass DEP & ASLR
  
  • And application specific bypasses/properties
Additional protections

- Raise vulnerability discovery and exploit development costs with additional protections
  
- The more line of defenses, the better!
  
- Implement protections against specific vulnerability classes and exploitation techniques
Additional protections

- Usually require source code changes (annotations) and/or recompilation of the application
  - To add run-time checks

- Implement safe datastructures and operations
  - E.g., heap manager, SEH, vtable
Additional protections

- **Stack canaries / Cookies**
  - Detects buffer overflows on stack
- **Heap protections**
  - Protects heap management data and operations
- **Pointer obfuscation**
- **GOT protection (BIND_NOW & RELRO)**
  - Relocate at load-time and mark GOT read-only
- **/GS (more than just cookies)**
- **/SAFESEH (link-time, provide list of valid handlers)**
- **SEHOP (run-time, registry, might cause compatibility issues, walk down SEH chain to final handler before dispatching / integrity check)**
- **Virtual Table Verification (VTV) & vtguard**
Stack canary / cookie

```c
void vulnFunc() {
    char buf[1024];
    read(STDIN, buf, 2048);
}
```

- Put an unknown value before local buffer
- Verify canary at function exit
- If value changed raise exception
Stack canary / cookie

- Detects linear buffer overflows on stack
  - At function exit

- Corruption of local stack not detected
  - Only if canary / cookie value is overwritten

- Incurs runtime overhead

- Effectiveness relies on secret
  - Leaking, predicting, guessing or brute-forcing might work in special cases
DEP & ASLR Adoption
## Windows

<table>
<thead>
<tr>
<th></th>
<th>Windows XP</th>
<th>Vista</th>
<th>Windows 7</th>
<th>Windows 8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DEP</strong></td>
<td>&gt; SP2 Opt-in</td>
<td>(Opt-in 32bit)</td>
<td>(Opt-in 32bit)</td>
<td>(Opt-in 32bit)</td>
</tr>
<tr>
<td><strong>ASLR</strong></td>
<td></td>
<td>Opt-in</td>
<td>Opt-in</td>
<td>Opt-in</td>
</tr>
<tr>
<td></td>
<td>Opt-in</td>
<td></td>
<td>Opt-in</td>
<td>Opt-in + Enforced by EXE + HE for 64bit + Full</td>
</tr>
</tbody>
</table>

* Opt-in for all images (/DYNAMICBASE)
  - Gaps might still exist, **not full ASLR** (E.g., VirtualAlloc or MapViewOfFile)
  - No difference between 32bit and 64bit ASLR

- Different ways to opt-in to ASLR and DEP
  - Linker flag, process creation attribute, SetProcessMitigationPolicy API, „MitigationOptions“ (Image File Execution Options), boot.ini switch

- **DEP is mandatory on 64-bit Windows**, on 32-bit Windows:
  - From Vista on, /NXCOMPAT linker switch sufficient
  - “Always On” can be configured system-wide (incl. exceptions list)
Windows ASLR

- Introduced with Vista

Windows 7 + Vista
- Gaps in ASLR might still exist
- Heaps and stacks randomized
- PEB/TEB randomized (limited entropy)
- VirtualAlloc and MapViewOfFile **not** randomized
- Non-ASLR images (without /DYNAMICBASE)
- Predictable memory regions (E.g., VirtualAlloc(), SharedUserData)

Windows 8
- Processes can force ASLR for non-ASLR images
- All bottom-up/top-down allocations randomized (opt-in, /DYNAMICBASE)
- More entropy for PEB/TEB
Windows 8 ASLR

- High entropy for 64-bit (8TB addr. Space)
  - High entropy bottom-up
    - stacks, heaps, mapped files, VirtualAlloc, etc.
    - Breaks spraying techniques
  - High entropy top-down
    - PEBs, TEBs, MEM_TOP_DOWN
- High entropy image randomization
  - /HIGHENTROPYVA (EXE)
Windows 8 / 8.1

- Further improvements
  - Sophisticated attacks still possible
  - Code-reuse & info leaks
  - Pwn2own 2014

“Writing exploits for Windows 8 will be very costly“

- 64bit, >IE10, VS 2012 + enable mitigations
Windows 8 / 8.1

- Built with enhanced /GS (v3, VS 2010)
  - Array index range checks (compiler-inserted)

- Sealed optimization
  - Eliminates indirect calls through vtable
  - Direct call faster and reduces attack surface

- vTable guard (class annotation required)
  - Random value “vtguard” at end of vtable
  - Verified before vtable gets used
  - IE10 uses it for a handful key classes
Windows 8 / 8.1

- ASLR improvements

- Windows heap improvements
  - Encounters specific exploitation techniques
    - LFH & integrity checks
  - Guard pages
    - Allocates guard pages between heap memory
    - Allocation order randomization (LFH, <16KB)

- Kernel improvements
  - DEP, ASLR, Kernel pool integrity checks
  - NULL dereference protection
Windows 7

- Ok, thanks, but I really just have 7...

- 64bit: DEP enforced
  - 32bit: Opt-in

- Enforce ASLR with EMET
  - And enable additional EMET hardening

- Use newest client software
  - Ideally, just one... and harden configuration

- And the usual: keep everything up-to-date!
Ubuntu Linux

<table>
<thead>
<tr>
<th></th>
<th>10.04</th>
<th>12.04</th>
<th>14.04</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX/DEP</td>
<td>(32bit, non-PAE: emulated)</td>
<td>(32bit, non-PAE: emulated)</td>
<td>(32bit, non-PAE: emulated)</td>
</tr>
<tr>
<td>ASLR</td>
<td>Full *</td>
<td>Full *</td>
<td>Full *</td>
</tr>
</tbody>
</table>

* Stack, libraries/mmap, brk, exec, vdso
  - System-wide \texttt{/proc/sys/kernel/randomize_va_space}

- Executables: ASLR only for PIE (position-independent executable)
  - On x86 this results in 5-10% performance loss
    - Therefore not all executables are compiled as PIE
  - On x86 64bit: PIE comes without penalties
    - But still not all executables are PIEs

- Shared libraries: use position-independent code by default

- NX/DEP requires PAE, otherwise emulated

https://wiki.ubuntu.com/Security/Features
http://en.wikipedia.org/wiki/Physical_Address_Extension
iOS

- ASLR introduced with iOS 4.3 in 2010
- For full ASLR, executables need to be PIE
  - Since 2011 default in XCode
  - Else (at least in iOS 4), exec, stack, linker are at fixed addresses
- Mandatory Code Signing
  - Application has to be signed (checked at execution time)
- Code Signing Enforcement (++)
  - Executed code has to be signed (checked at runtime)
  - No new executable code can be generated, mprotect(addr,len,PROT_EXEC)
  - Mapped code is not mutable (W^X enforcement)

<table>
<thead>
<tr>
<th></th>
<th>iOS 5</th>
<th>iOS 6</th>
<th>iOS 7</th>
<th>iOS 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>NX/DEP</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>ASLR</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
* Stack and libraries/mmap
i.e. no ASLR for executable, heap/brk, linker

- NX/DEP introduced with 2.3 (Gingerbread)
- Partial ASLR introduced with 4.0
  - Full ASLR and PIE support since 4.1
    - Stack, libraries/mmap, executable, heap/brk, linker

https://www.duosecurity.com/blog/exploit-mitigations-in-android-jelly-bean-4-1
https://www.duosecurity.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0
Advanced Attacks

```
* Welcome to CityPower

Login:

nmap -v -sS -O 10.2.2.2
Starting nmap V. 2.54BETA25
Insufficient responses for TCP sequencing (3), OS detection
accurate
Interesting ports on 10.2.2.2:
(The 1539 ports scanned but not shown below are in state: open)
Port State Service
1022/tcp open 闭环
No exact OS matches for host

Nmap run completed -- 1 IP address (1 host up) scanned
Connecting to 10.2.2.2, rootpw="Z10N0101"
Root accessing to exploit SSH/1 CRC32 ... successful.
System open: Access Level <9>
```

```
root@10.2.2.2's password: 
Access Granted
```
Application specific attacks

- **Today, attacks are much more application and vulnerability specific!**
  - No universal exploitation techniques
    - Attackers focus on promising applications
  - ASLR bypass depends on situation
    - Or uses information leaks
  - DEP bypassed by code-reuse techniques
    - ROP chain depends on available gadgets
Heap-spraying

- Payload delivery technique
  - Shellcode and/or ROP chain

- Attacker goal: deliver data to predictable address on heap

- First documented usage in 2001
  - Widespread use in browser exploits since 2005

http://en.wikipedia.org/wiki/Heap_spraying
Heap-spraying

- Only possible if attacker has (partial) control over heap allocations
  - Data and layout (indirectly)

- Popular in applications with client-side scripting support
  - E.g., Browsers, Flash, PDF Reader

- OS, heap implementation and application specific
Heap-spraying

Victim

Stack

Heap

Code

Attacker

Payload address unknown

? 

<html>
<script>
var shellcode = unescape('%u4141%u4141');
</script>
</html>
Heap-spraying

Script sprays objects/data to the heap!

```html
<html>
<script>
var shellcode = unescape('%u4141%u4141');
var bigblock = unescape('%u9090%u9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) 
bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x40000) 
block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){
  memory[i] = block + shellcode 
}
</script>
</html>
```
Heap-spraying

Payload is at **known address** now!
Heap-spraying

- Use control over heap to spray the heap with data (payload)
  - After spraying the data is found at a deterministic address
- Sprayed data can be anything
  - E.g., JS objects/strings, images

http://en.wikipedia.org/wiki/Heap_spraying
Heap-spraying

- Note: not feasible for effective ASLR bypass!
  - Entire address space too large to spray
    - Especially for 64bit address space
  - Heap base address range has to be non-randomized
    - E.g., VirtualAlloc no ASLR up to Windows 7

- If effective ASLR in place heap-spraying can still be useful
  - Spray heap and corrupt heap relative offset
Heap-spraying

- Without DEP: heap is executable

<table>
<thead>
<tr>
<th>nop sled</th>
<th>shellcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop sled</td>
<td>shellcode</td>
</tr>
<tr>
<td>nop sled</td>
<td>shellcode</td>
</tr>
<tr>
<td>nop sled</td>
<td>shellcode</td>
</tr>
<tr>
<td>nop sled</td>
<td>shellcode</td>
</tr>
<tr>
<td>nop sled</td>
<td>shellcode</td>
</tr>
</tbody>
</table>

%eip

http://en.wikipedia.org/wiki/Heap_spraying
Heap-spraying

- With DEP (exact spray desirable)

1. | padding | ROP chain | shellcode |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%esp = make %esp point to heap spray, %eip to ret</td>
<td></td>
</tr>
</tbody>
</table>

2. | padding | ROP chain | shellcode |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%esp</td>
<td>%eip</td>
</tr>
<tr>
<td>ROP chain:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make heap RWX +</td>
<td></td>
<td></td>
</tr>
<tr>
<td>transfer control to shellcode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. | ROP nops | ROP chain | shellcode |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%esp</td>
<td></td>
</tr>
</tbody>
</table>
Heap-spraying countermeasures

- Heap-spray detection inside browsers
  - Detect suspicious allocations
    - Patterns, valid instructions, large static blocks
    - Nozzle (>IE9) and BuBBle (>Firefox 9)

- Pre-allocation of popular regions
  - EMET or HeapLocker, e.g., 0x0c0c0c0c0c

- Monitor memory usage and limit amount of memory per process / script

Heap-spraying variations

- Randomize bytes in spray
- Don't use strings
  - DEPS: „DOM Element Property Spray“
  - DOM object properties = payload
- HTML5 spraying
  - Uses canvas objects for payload and web workers for speed up

http://exploiting.wordpress.com/2012/10/03/html5-heap-spray-eusecwest-2012/
Enforcing information leaks

• To bypass ASLR attackers can “construct” memory leaks

  • Corrupt length field of an object and read object data, e.g., JS string

  • Length field corruption has to be feasible without bypassing ASLR

    • Use heap-spraying for reliability

Pointer inference

- Recover addresses of internal objects without explicit memory leaks
  - Only by „interacting“
  - Like a side-channel attack

- E.g., over ActionScript Dictionaries as described by Dion Blazakis
  - Associative map data structure
  - Keys can be integers, strings, other objects
  - Hashtable uses values or references
  - By placing ordered values into the data structure and iterating through it bits of the object's address are disclosed
JITs

- **JIT: Just-In-Time**
  - Refers to execution engines / compilers
  - Native just-in-time code generation
  - E.g., javascript engines in browsers, Java VMs, .NET CLR, ActionScript, other language runtimes
- **JIT types: Method & Tracing JITs**
  - Front-end, syntax parser, produces IR
  - Back-end, generates native code out of IR
JITs

- JIT vulnerabilities
  - Incorrect code generation
    - Like having a bug in generated native code
  - Logic errors (in generated code)
  - Information leaks
  - Diversion of control-flow

- JITs are sources of vulnerabilities and means of exploitation
JIT-spraying

- Remember DEP?
  - Makes machine-code injection difficult
- But wait... JITs generate code!
  - They need RWX memory (code cache)
  - Native code is generated out of untrusted attacker supplied higher-level source or code, e.g., javascript or Java bytecode
JIT-spraying

- Generate predictable byte sequences in generated native code

- Example from “Attacking Clientside JIT Compilers”

```javascript
var a, b, c, d = -6.828527034422786e-229;
```

Floating point value uses 64bit in 32bit x86, will be represented as 0x9090909090909090

```
movl $0x90909090,0x5c0(%esi)
movl $0x90909090,0x5c4(%esi)
movl $0x90909090,0x5c8(%esi)
movl $0x90909090,0x5cc(%esi)
movl $0x90909090,0x5d0(%esi)
movl $0x90909090,0x5d4(%esi)
movl $0x90909090,0x5d8(%esi)
movl $0x90909090,0x5dc(%esi)
```

0x90 is opcode for nop

x86 has variable instructions length, jump into valid instructions possible
JIT-spraying

- If JIT memory locations are predictable
  - Spraying or non-randomized allocations
  - Information leaks

- Attackers might inject code into executable memory and divert execution to it
  - Or spray gadgets throughout JIT memory
JIT hardening

- Randomization / full ASLR
- Page permissions
  - RW for generation, RX for execution
- Guard pages
  - Prevent overflows from RW to RWX pages
  - Overwrite generated native code and trigger execution
- Constant folding / blinding
- Allocation size restrictions for native code
- Random NOP insertion, random code base offsets
Attacking safe language VMs

• Just use a type & memory safe language?

But language VM

• May be implemented in an unsafe language
• May use or provide interfaces to unsafe libraries

Exploit memory errors in the VM or in unsafe libraries used by the VM or the application
Java VM written in C/C++
Java VM written in C/C++

Java Application Process

Software Stack

Java Application

Java API & Libraries

Execution Engine – JIT | GC

JNI

Java VM

Libraries

Operating System

Hardware

Potentially prone to memory errors & corruption
Attacking safe language VMs

E.g., Java VM

- CVE-2013-1491
- Affected Oracle Java SE 7 / 6 / 5
- Memory error in OpenType Fonts handling within native layer of JRE
  - Leveraged to arbitrary code execution
  - Completely bypassed DEP & ASLR

Demonstrated at Pwn2Own at CanSecWest 2013 by Joshua Drake (on Windows 8 + Java SE 7 Update 17)
http://www.accuvant.com/blog/pwn2own-2013-java-7-se-memory-corruption
Attacking Java VM

Attack surface

- Any Java application relying on native code
- Untrusted Java Applet (over the web)
  - Java Applet under attacker control
  - Sandboxed, but a lot of native code reachable
    - Image, sound, data processing

Demonstrated at Pwn2Own at CanSecWest 2013 by Joshua Drake (on Windows 8 + Java SE 7 Update 17)
http://www.accuvant.com/blog/pwn2own-2013-java-7-se-memory-corruption
Oracle JRE 6 / 7 / 8

- **JRE 6**
  - Only used /GS and /SafeSEH (stack protections)
  - No DEP or ASLR
    - DEP could be enforced
    - But msvcr71.dll allowed easy bypass (no ASLR)
      - Shipped with all releases of JRE 6

- **JRE 7 & 8**
  - Many improvements
  - DEP & ASLR enabled

Additional hardening

- A lot of additional hardening techniques
  - EMET on Windows
  - PaX/grsecurity patches on Linux
  - Hardened configurations
  - And many additional tools and techniques
Enhanced Mitigation Experience Toolkit

• EMET 5.0, released 31.7.2014
  • 3.x 2012, 4.x 2013

• Hardening Toolkit
  • Set of protections against exploitation techniques
  • Works on binaries, configurable per process
  • Interesting for legacy software

• Supports enterprise deployment
  • Built-in Group Policy and System Center Configuration Manager support

EMET 5.0

- Application compatibility risk
  - Testing required before production rollout

- Some features also exist without EMET
  - But can be activated for older Windows versions

- “EMET User's Guide“ not that detailed


EMET 5.0

- SEHOP - run-time validation of SEH chain
- DEP enforcement (without flag)
- Heapspray pre-allocation
- ASLR enforcement (without flag)
  - < Windows 8 (native ASLR)
- EAF – Export Address Table Access Filtering
- EAF+
  - Stack register boundary check, stack/frame pointer mismatch
  - Detects memory read accesses to certain tables/headers
EMET 5.0

- Bottom-up randomization
  - Randomizes (8 bits entropy) base address of bottom-up allocations (heaps, stacks, other memory allocations)

- ROP mitigations (all for 32bit and **some for 64bit**)
  - Load library checks (LoadLibrary API, UNC paths -> network)
  - Memory protection check, disallows making stack executable
  - Caller checks for critical functions
    - Check if transfer originated from call instruction
  - Simulate execution flow
    - Check after a critical function if ROP chain is executed
  - Stack pivot
    - Check if stack has been pivoted
EMET 5.0

- **Attack Surface Reduction**
  - DLL blacklist per application (and Security Zone)
  - E.g., disable Java plugin within IE in Internet Zone

- **Advanced Mitigations for ROP**
  - Deep hooks (protect critical APIs on all levels)
  - Anti detours
  - Banned functions (ntdll!LdrHotPatchRoutine)
EMET 5.0
# EMET 5.0

## Incompatible mitigations

<table>
<thead>
<tr>
<th>Product</th>
<th>EMET 4.1 Update 1</th>
<th>EMET 5.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-Zip Console/GUI/File Manager</td>
<td>EAF</td>
<td>EAF</td>
</tr>
<tr>
<td>Adobe Acrobat</td>
<td>Not applicable</td>
<td>EAF+ (AcroRd32.dll)</td>
</tr>
<tr>
<td>Adobe Reader</td>
<td>Not applicable</td>
<td>EAF+ (AcroRd32.dll)</td>
</tr>
<tr>
<td>Certain AMD/ATI video drivers</td>
<td>System ASLR=AlwaysOn</td>
<td>System ASLR=AlwaysOn</td>
</tr>
<tr>
<td>DropBox</td>
<td>EAF</td>
<td>EAF</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>SEHOP*</td>
<td>SEHOP*</td>
</tr>
<tr>
<td>Google Talk</td>
<td>DEP, SEHOP*</td>
<td>DEP, SEHOP*</td>
</tr>
<tr>
<td>Immidio Flex+</td>
<td>Not applicable</td>
<td>EAF</td>
</tr>
<tr>
<td>Microsoft Office Web Components (OWC)</td>
<td>System DEP=AlwaysOn</td>
<td>System DEP=AlwaysOn</td>
</tr>
<tr>
<td>Microsoft Word</td>
<td>Headspray</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Oracle Java</td>
<td>Heapspray</td>
<td>Heapspray</td>
</tr>
<tr>
<td>Skype</td>
<td>EAF</td>
<td>EAF</td>
</tr>
<tr>
<td>SolarWinds Syslogd Manager</td>
<td>EAF</td>
<td>EAF</td>
</tr>
<tr>
<td>VLC Player 2.1.3+</td>
<td>SimExecFlow</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Windows Media Player</td>
<td>MandatoryASLR, EAF, SEHOP*</td>
<td>MandatoryASLR, EAF, SEHOP*</td>
</tr>
<tr>
<td>Windows Photo Gallery</td>
<td>Caller</td>
<td>Caller</td>
</tr>
</tbody>
</table>

* Only in Windows Vista and earlier versions
EMET 4.1 bypass

- From the “Executive Summary”
  
  “We were able to bypass EMET’s protections in example code and with a real world browser exploit.”

- No ROP protections for 64 bit processes
- EMET raises the costs for exploit development
  - But no magic bullet

“Bypassing EMET 4.1” Jared DeMott, Security Researcher, Bromium
Linux - PaX

- Better ASLR implementation
- Additional kernel protections
- No RWX pages (basically W^X)
- Random padding between thread stacks
- Hardened BPF JIT in kernel
- Exploit brute-force protection
  - Restrict forks on network services
Linux - Ubuntu

- Long list of security features
  - Run-time and compile-time hardening

- Also support for MAC (instead of DAC)
  - Implemented as LSM:
    AppArmor, SELinux, SMACK

- SECCOMP: syscall filtering

https://wiki.ubuntu.com/Security/Features
Sandboxing

• Any mean of isolation & reducing privileges
  • Goal: restrict potentially malicious code
    • Not trusted or not trustworthy

• Can be implemented on any layer
  • Can come with virtualization/emulation

• App sandbox on iOS / Android
• A virtual machine
• Browser process sandbox
Conclusion
Conclusion

• Memory errors are still an issue

• Attacks and defenses gain complexity

• In the end... there is still a residual risk
Conclusion

- There are a lot of hardening technologies
  - That just have to be used
  - Proper risk assessment and technical understanding is key

- There is a lot of vendor awareness
  - Consequently, products are getting more secure

- New effective technologies are on the rise...
Thanks!

antonio.barresi@inf.ethz.ch