wiredmikey writes with a snippet from Security Week:

"Much of the talk about cybercrime remains focused on East Asia. But according to a new report, it is hackers in Eastern Europe that have actually emerged as more sophisticated. In a report entitled 'Peter the Great vs. Sun Tzu' ... compared hackers from the two regions. His conclusion — the Eastern Europeans are far more insidious and strategic. While East Asian groups tend to work for other organizations interested in their skills, hackers from Eastern Europe generally operate in small, independent units, and are focused on profit. Their infrastructure tends to be developed by them specifically for their own use in attacks. 'They [Eastern European groups] tend to want to be in control of their entire infrastructure and will routinely set up their own servers for use in attacks, develop their own DNS servers to route traffic and create sophisticated traffic directional systems used in their attacks,' according to the report. 'If they do go outside, they will carefully select bulletproof hosts to support their infrastructure. It is their hallmark to maintain control of the whole stack similar to the business models pioneered by Apple.'"
Low-level software vulnerability protection mechanisms

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu
How can we help prevent exploitation of buffer overflows and other control flow hijacking?

- Non-executable memory pages
- Return-into-libc exploits, Return-oriented programming
- Address space layout randomization
- StackGuard, StackShield
- Software fault isolation
Process memory layout

.text
  machine code of executable
.data
  global initialized variables
.bss
  "below stack section"
global uninitialized variables

.heap
  dynamic variables
.stack
  local variables, track func calls
.Env.
  environment variables, arguments to program

Unused space

Low memory addresses

High memory addresses
Typical return ptr overwrite exploit

Why should the machine interpret stack data as instructions?
$W^X \ (W \ xor \ X)$

- The idea: mark memory page as either
  - Writable or Executable (not both)
- Specifically: make heap and stack non-executable
$W^X \ (W \ xor \ X)$

- **AMD64**: NX bit (Non-Executable)
- **IA-64**: XD bit (eXecute Disabled)
- **ARMv6**: XN bit (eXecute Never)
  - Extra bit in each page table entry
  - Processor refuses to execute code if bit = 1
  - Mark heap and stack segments as such
$W^X \ (W \text{ xor } X)$

Software emulation of NX bits

- ExecShield (RedHat Linux)
- PaX (Page-eXec) (uses NX bit if available)

`mprotect()`

- Process can set permissions on memory pages
Will W^X stop:

AlephOne’s stack overflow exploit? Yes

Stack smash that overwrites pointer to point at shell code in Heap or Env variable? Yes

Heap overflow with same shell location? Yes

Double free with same shell location? Yes
Limitations of W^X

Software emulation ... 
• May not be perfect and is slow
• E.g., double-free or format-string vulnerability may allow turning off protections

Breaking compatibility
• GCC stack trampolines (calling conventions, nested functions)
• Just-in-time (JIT) compilation using heap

Exploits designed to only run existing code
What about circumventing $W^X$?

Exploits designed to only run existing code
Return-into-libc exploits

• libc is standard C library, included in all processes
• system() --- execute commands on system
Return-into-libc exploits

Overwrite EIP with address of system() function

junk2 just some filler: returned to after system call

first argument to system() is ptr to “/bin/sh”
Return-into-libc exploits

This simple exploit has a few deficiencies (from attacker’s pov):
- Crashes after exiting called /bin/sh ( easy to fix with exit() )
- system() drops privileges by default

```
wrap.c :
main() {
  setuid(0);
  setgid(0);
  system("bin/sh");
}
```

```
printf("%3$n", ... )
%3n means “write number of bytes in format string up to the format token into third parameter”
```

```
execl("./wrap", "./wrap", 0 )
```

```
 escrit 0 here
```
Return-into-libc exploits

These exploits only execute instructions marked executable

W^X cannot stop such an attack

<table>
<thead>
<tr>
<th>junk</th>
<th>addr of printf</th>
<th>addr of <code>execl</code></th>
<th>addr of <code>%3\$n</code></th>
<th>addr of <code>./wrap</code></th>
<th>addr of <code>./wrap</code></th>
<th>addr of HERE</th>
</tr>
</thead>
</table>

wrap.c :
```c
main() {
  setuid(0);
  setgid(0);
  system("bin/sh");
}
```

`execl( "./wrap", "./wrap", 0 )`

`printf( "%3$n", ... )`

%3$n means “write number of bytes in format string up to the format token into third parameter”
Return-into-libc exploits

Return-into-libc may seem limited:
- Only useful for calling libc functions
- Okay in last example, but not always sufficient
- Before W^X, exploit could run arbitrary code

Can we not inject any malicious code and yet have an exploit that runs arbitrary code?
Return-oriented programming (ROP)

Second return-into-libc exploit:
  self-modifying exploit buffer to call a sequence of libc calls

Logical extreme:
  chain together a long sequence of calls to code

But we want arbitrary code, not sequence of libc calls:
  chain together a long sequence of calls to code snippets
Return-oriented programming (ROP)

Figure 2: Load the constant Oxdeadbeef into %edx.

From Shacham “The Geometry of Innocent Flesh on the Bone...” 2007

If this is on stack and (*) is return pointer after buffer overflow, then the result will be loading Oxdeadbeef into edx register
Return-oriented programming (ROP)

Figure 3: Load a word in memory into %eax.

From Shacham “The Geometry of Innocent Flesh on the Bone...” 2007
From Shacham
“The Geometry of Innocent Flesh on the Bone...” 2007
ROP where do we get code snippets?

Buchanan et al., Blackhat 2008
W^X wrapup

W^X does not prevent arbitrary code execution, but does make it harder!

What else can we do?
Address space layout randomization (ASLR)

dynamically linked libraries (libc) go in here

Low memory addresses

High memory addresses
Address space layout randomization (ASLR)

dynamically linked libraries (libc) go in here

.random
16-bit
offset

Randomize offsets of three areas
16 bits, 16 bits, 24 bits of randomness

Adds unpredictability... but how much?

PaX implementation for example:

- Randomize offsets of three areas
- 16 bits, 16 bits, 24 bits of randomness
- Adds unpredictability... but how much?
Defeating ASLR

• W^X not on?
  Large nop sled with classic buffer overflow

• Use a vulnerability that can be used to leak address information (e.g., printf arbitrary read)

• Brute force the address
Defeating ASLR

Brute-forcing example from reading “On the effectiveness of Address Space Layout Randomization” by Shacham et al.

Apache web server with Oracle 9 PL/SQL module

Apache forks off child process to handle request

There is a buffer overflow in module that helps process request

request

response
Defeating ASLR

Brute-forcing example from reading “On the effectiveness of Address Space Layout Randomization” by Shacham et al.

Attacker makes a guess of where usleep() is located in memory

Failure will crash the child process immediately and therefore kill connection

Success will crash the child process after sleeping for 0x01010101 microseconds and kill connection

<table>
<thead>
<tr>
<th>Maximum time (s)</th>
<th>Average time (s)</th>
<th>minimum time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>810</td>
<td>216</td>
<td>29</td>
</tr>
</tbody>
</table>
If on 64-bit architecture, randomization significantly more effective

Can also randomize more stuff:
• Instruction set randomization
• per-memory-allocation randomization
• etc.
Protecting the stack

Can we protect the return address from being overwritten?

Two approaches:
- Detect manipulation (and then fail safe)
- Prevent it completely
Detection: stack canaries

Canary value can be:
- Random value (choose once for whole process)
- NULL bytes / EOF / etc. (string functions won’t copy past canary)

On end of function, check that canary is correct, if not fail safe
Detection: stack canaries

StackGuard:
• GCC extension that adds runtime canary checking
• 8% overhead on Apache

ProPolice:
• Modifies how canaries inserted
• Adds protection for registers
• Sorts variables so arrays are highest in stack
Detection: stack canaries

Low memory addresses

High memory addresses

Discussion: How would you get around it?

Detection: copying values to safe location

StackShield:
• Function call: copy return address to a safe location (beginning of .data)
• Check if stack value is different on function exit

Discussion: How would you get around this?
Prevention

StackGhost:
• Encrypting the return address
  • XOR with random value on function entrance
  • XOR with same value on function exit
• Per-kernel XOR vs. Per-process XOR
• Return address stack
Confinement (sand boxing)

- All the mechanisms thus far are circumventable
- Can we at least confine code that is potentially vulnerable so it doesn’t cause harm?
Simple example is chroot

   chroot /tmp/guest
   su guest

Now all file access are prepended with /tmp/guest

   open("/etc/passwd", "r")

Attempts to open /tmp/guest/etc/passwd

Limitation is that all needed files must be inside chroot jail

   jailkit

Limitation: network access not inhibited

Limitation: must not allow programs in jail to get get root privileges
Escaping jails

```python
open("../../etc/passwd", "r")
```
Attempts to open
```
/tmp/guest/../../etc/passwd
```

chroot should only be executable by root

```bash
create /aaa/etc/passwd
chroot /aaa
su root
```
System call interposition

• Malicious code must make system calls in order to do bad things
• So monitor system calls!
Figure 1. System Call Interposition in Janus

Diagram from Garfinkel 2003
Software-fault isolation example: Google Native Client

Goal: run native code from a web browser safely

Examples are Quake and XaoS ported over

From Yee et al. 2009

Figure 1: Hypothetical NaCl-based application for editing and sharing photos. Untrusted modules have a grey background.
Software-fault isolation example: Google Native Client

Inner sandbox

- static analysis to detect flaws
- require code to abide by alignment and structure rules, allowing disassembly.
- Find any disallowed machine instructions
- x86 segmented memory to confine data and instruction references

Validator quickly checks that a binary abides by these rules
Software-fault isolation example: Google Native Client

Outer sandbox
• system call interposition to monitor
• similar to Janus / ptrace
## Native client spec perf

<table>
<thead>
<tr>
<th>Program</th>
<th>static</th>
<th>aligned</th>
<th>NaCl</th>
<th>increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>200</td>
<td>203</td>
<td>203</td>
<td>1.5%</td>
</tr>
<tr>
<td>art</td>
<td>46.3</td>
<td>48.7</td>
<td>47.2</td>
<td>1.9%</td>
</tr>
<tr>
<td>bzip2</td>
<td>103</td>
<td>104</td>
<td>104</td>
<td>1.9%</td>
</tr>
<tr>
<td>crafty</td>
<td>113</td>
<td>124</td>
<td>127</td>
<td>12%</td>
</tr>
<tr>
<td>eon</td>
<td>79.2</td>
<td>76.9</td>
<td>82.6</td>
<td>4.3%</td>
</tr>
<tr>
<td>equake</td>
<td>62.3</td>
<td>62.9</td>
<td>62.5</td>
<td>0.3%</td>
</tr>
<tr>
<td>gap</td>
<td>63.9</td>
<td>64.0</td>
<td>65.4</td>
<td>2.4%</td>
</tr>
<tr>
<td>gcc</td>
<td>52.3</td>
<td>54.7</td>
<td>57.0</td>
<td>9.0%</td>
</tr>
<tr>
<td>gzip</td>
<td>149</td>
<td>149</td>
<td>148</td>
<td>-0.7%</td>
</tr>
<tr>
<td>mcf</td>
<td>65.7</td>
<td>65.7</td>
<td>66.2</td>
<td>0.8%</td>
</tr>
<tr>
<td>mesa</td>
<td>87.4</td>
<td>89.8</td>
<td>92.5</td>
<td>5.8%</td>
</tr>
<tr>
<td>parser</td>
<td>126</td>
<td>128</td>
<td>128</td>
<td>1.6%</td>
</tr>
<tr>
<td>perlbmk</td>
<td>94.0</td>
<td>99.3</td>
<td>106</td>
<td>13%</td>
</tr>
<tr>
<td>twolf</td>
<td>154</td>
<td>163</td>
<td>165</td>
<td>7.1%</td>
</tr>
<tr>
<td>vortex</td>
<td>112</td>
<td>116</td>
<td>124</td>
<td>11%</td>
</tr>
<tr>
<td>vpr</td>
<td>90.7</td>
<td>88.4</td>
<td>89.6</td>
<td>-1.2%</td>
</tr>
</tbody>
</table>

Table 4: SPEC2000 performance. Execution time is in seconds. All binaries are statically linked.
Native client Quake perf

<table>
<thead>
<tr>
<th>Run #</th>
<th>Native Client</th>
<th>Linux Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>143.2</td>
<td>142.9</td>
</tr>
<tr>
<td>2</td>
<td>143.6</td>
<td>143.4</td>
</tr>
<tr>
<td>3</td>
<td>144.2</td>
<td>143.5</td>
</tr>
<tr>
<td>Average</td>
<td>143.7</td>
<td>143.3</td>
</tr>
</tbody>
</table>

Table 8: Quake performance comparison. Numbers are in frames per second.
More sandboxing: virtualization

- Modern virtual machines (VMs) often used for sandboxing

[Diagram showing a two-tiered virtual machine structure with labeled components: VM monitor, Host OS, Hardware, Guest OS, App]
More sand boxing: virtualization

• Malicious use of virtualization: blue pill virus
Discussion: state of low level software security

• Do you think Native Client is fool proof?
  – Winner of native client competition 2009
  – vulns found in validator,

• What about VM-based sandboxing?
  – Check for escape-from-VM vulnerabilities (hypervisor or management guest VM)

• How does all this make you feel?