Acknowledgments: Some of the slides are fully or partially obtained from other sources. Reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.
Run-Time protection/enforcement

• In many instances we only have access to the binary
• How do we analyze the binary for vulnerabilities?
• How do we protect the binary from exploitation?
• This would be our topic for the next few lectures
Why Binary Code?

- Access to the source code often is not possible:
  - Proprietary software packages
  - Stripped executables
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
- Binary code is the only authoritative version of the program
  - Changes occurring in the compile, optimize and link steps can create non-trivial semantic differences from the source and binary
- Worms and viruses are rarely provided with source code
Binary Analysis and Editing

- **Analysis**: processing of the binary code to extract syntactic and symbolic information
  - Symbol tables (if present)
  - Decode (disassemble) instructions
  - Control-flow information: basic blocks, loops, functions
  - Data-flow information: from basic register information to highly sophisticated (and expensive) analyses
Binary Analysis and Editing

- **Binary rewriting**: static (before execution) modification of a binary program
  - Analyze the program and then insert, remove, or change the binary code, producing a new binary

- **Dynamic instrumentation**: dynamic (during execution) modification of a binary program
  - Analyze the code of the running program and then insert, remove, or change the binary code, changing the execution of the program
  - Can operate on running programs and servers
Uses of Binary Analysis and Editing

- Cyber-forensics
  - Analysis: understand the nature of malicious code
  - Binary-rewriting: produce a new version of the code that might be instrumented, sandboxed, or modified for study
  - Dynamic instrumentation: same features, but can do it interactively on an executing program
  - Hybrid static/dynamic: control execution and produce intermediate versions of the binary that can be re-executed (and further instrumented)
- Program tracing: instructions, memory accesses, function calls, system calls, . . .
- Debugging
- Testing, Performance profiling Performance modeling
- Reverse engineering
Binary patch

Application
Program

Function foo

Trampoline

Pre-Instrumentation

Relocated Instruction(s)

Post-Instrumentation

pop ecx; puts the return address to ecx
jmp ecx; jumps to the return address

After Patch:

pop ecx; puts the return address to ecx
cmp ecx, 0x08048456; check that we return to the right place
jne 0x41414141; crash
jmp ecx; effectively return
Dynamic Binary Instrumentation

• A DBI is a way to execute an external code before or/and after each instruction/routine

• With a DBI you can:
  • Analyze the binary execution step-by-step
    • Context memory
    • Context registers
  • Only analyze the executed code
Available Tools

• Binary re-writing:
  • e.g.: Alto, Vulcan, Diablo, etc.

• Binary Instrumentation:
  • e.g. PIN, Valgrind, DynInst, etc
Motivation

• Worms exploit several software vulnerabilities
  • buffer overflow
  • “format string” vulnerability
• Attack detectors ideally should:
  • Detect new attacks and detect them early
  • Be easy to deploy
  • Few false positives and false negatives
  • Be able to automatically generate filters and sharable fingerprints

[Papadopoulos’11]
Motivation (contd.)

• Attack detectors are:
  • Coarse grained detectors
    • Detect anomalous behavior but do not provide detailed information about the vulnerability
  • Scan detectors, anomaly detectors
  • Fine grained detectors are highly desirable
    • Detect attacks on programs vulnerabilities and hence provide detailed information about the attack
    • But some require source code (typically not available for commercial software), recompilation, bounds checking, library recompilation, source code modification, etc.
  • Other options: content-based filtering (e.g., IDS’ such as snort and Bro), but automatic signature generation is hard
TaintCheck: Basic Ideas

• Program execution normally derived from trusted sources, not attacker input
• Mark all input data to the computer as “tainted” (e.g., network, stdin, etc.)
• Monitor program execution and track how tainted data propagates (follow bytes, arithmetic operations, jump addresses, etc.)
• Detect when tainted data is used in dangerous ways

[Papadopoulos’11]
Step 1: Add Taint Checking code

- TaintCheck first runs the code through an emulation environment (Valgrind) and adds instructions to monitor tainted memory.
TaintCheck Detection Modules

- **TaintSeed**: Mark untrusted data as tainted
- **TaintTracker**: Track each instruction, determine if result is tainted
- **TaintAssert**: Check if tainted data is used dangerously
  - Jump addresses: function pointers or offsets
  - Format strings: is tainted data used as a format string arg?
  - System call arguments
  - Application or library customized checks

*Figure 1. TaintCheck detection of an attack. (Exploit Analyzer not shown).*
TaintSeed

• Marks any data from untrusted sources as “tainted”
  • Each byte of memory has a four-byte shadow memory (ick!) that stores a pointer to a Taint data structure if that location is tainted
    • records the system call number, a snapshot of the current stack and a copy of the data that was written.
  • Else store a NULL pointer

Memory is mapped to TDS
TaintTracker

- Tracks each instruction that manipulates data in order to determine whether the result is tainted.
  - When the result of an instruction is tainted by one of the operands, TaintTracker sets the shadow memory of the result to point to the same Taint data structure as the tainted operand.

Memory is mapped to TDS

Result is mapped to TDS

[Papadopoulos’11]
TaintAssert

- Checks whether tainted data is used in ways that its policy defines as illegitimate

Memory is mapped to TDS → Operand is mapped to TDS → vulnerability

[1] Papadopoulos’11
TaintCheck Operation

- Taint seed
- Memory byte
- Shadow Memory
- TaintTracker
- Untainted
- TaintData structure*
- TaintCheck

* TDS holds the system call number, a snapshot of the current stack, and a copy of the data that was written

Exploit Analyzer

- Use as Fn pointer
- Attack detected

Spring 1398
Ce 874 - Taint Analysis
Papadopoulos’11
Exploit Analyzer

• Provides useful information about how the exploit happened, and what the exploit attempts to do
• Useful to generate exploit fingerprints

• Usage:
  • Identifying vulnerabilities.
  • Generating exploit signature.
Dynamic Taint Analysis

- Jump addresses:
  - Checks whether tainted data is used as a jump target
  - Instrument before each Ucode jump instruction
- Format strings:
  - Checks whether tainted data is used as format string argument
  - Intercept calls to the printf family of functions
- System call arguments:
  - Checks whether the arguments specified in system calls are tainted
  - Optional policy for execv system call
- Application or library-specific checks:
  - To detect application or library specific attacks
When does TaintCheck Fail?

- A false negative occurs if an attacker can cause sensitive data to take on a value without that data becoming tainted
  - E.g. if \( x == 0 \) \( y = 0 \); else if \( x == 1 \) \( y = 1 \); ...
- If values are copied from hard-coded literals, rather than arithmetically derived from the input
  - IIS translates ASCII input into Unicode via a table
- If TaintCheck is configured to trust inputs that should not be trusted
  - data from the network could be first written to a file on disk, and then read back into memory
When does TaintCheck give a False Positive?

• TaintCheck detects that tainted data is being used in an illegitimate way even when there is no attack taking place. Possibilities:
  • There are vulnerabilities in the program and need to be fixed, or
  • The program performs sanity checks before using the data

[Papadopoulos’11]
\[ y = x + 42 \]

... goto y

Input is tainted

TaintSeed

\[ \Delta \]

<table>
<thead>
<tr>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Var</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau )</td>
<td></td>
</tr>
</tbody>
</table>
\( x = \text{get\_input}() \)

\( y = x + 42 \)

\( \text{goto } y \)

Data derived from user input is tainted

TaintTracker

\begin{tabular}{|c|c|}
\hline
Var & Val \\
\hline
x & 7 \\
\hline
y & 49 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
Var & Tainted? \\
\hline
x & T \\
\hline
y & T \\
\hline
\end{tabular}

[Brumley’10]
\( x = \text{get_input}() \)
\( y = x + 42 \)

\[ \begin{array}{c|c}
\text{Var} & \text{Val} \\
\hline
x & 7 \\
y & 49 \\
\end{array} \]

\[ \begin{array}{c|c}
\text{Var} & \text{Tainted?} \\
\hline
x & T \\
y & T \\
\end{array} \]

Policy Violation Detected

TaintAssert

[Brumley’10]
Spring 1398
Ce 874 - Taint Analysis

x = get_input( )

y = ...

... goto y

Jumping to overwritten return address

... strcpy(buffer,argv[1])
... return ;

[Brumley’10]
### Memory Load

#### Variables

<table>
<thead>
<tr>
<th>Δ</th>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>7</td>
</tr>
</tbody>
</table>

#### Tainted?

<table>
<thead>
<tr>
<th>τ</th>
<th>Var</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>T</td>
</tr>
</tbody>
</table>

#### Memory

<table>
<thead>
<tr>
<th>μ</th>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>τ_μ</th>
<th>Addr</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>F</td>
</tr>
</tbody>
</table>

[Brumley’10]
Problem: Memory Addresses

\[ x = \text{get\_input}( ) \]
\[ y = \text{load}( ) \]
\[ \ldots \]
\[ \text{goto } y \]

All values derived from user input are tainted??

<table>
<thead>
<tr>
<th>( \Delta )</th>
<th>Var</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \mu )</th>
<th>Addr</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \tau_\mu )</th>
<th>Addr</th>
<th>Tainted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Policy 1: Taint depends only on the memory cell

Undertainting
Failing to identify tainted values
- e.g., missing exploits
Policy 2: If either the address or the memory cell is tainted, then the value is tainted.

Overtainting

Unaffected values are tainted - e.g., exploits on safe inputs.
General Challenge

- State-of-the-Art is not perfect for all programs

Undertainting: Policy may miss taint

Overtainting: Policy may wrongly detect taint
Compatibility with Existing Code

• Does TaintCheck raise false alerts?
• Networked programs: 158K+ DNS queries
  • No false +ves
• All (!!) client and non-network programs (tainted data is stdin):
  • Only vim and firebird caused false +ves (data from config files used as offset to jump address)
## Attack Detection: Synthetic + Actual Exploits

<table>
<thead>
<tr>
<th>Program</th>
<th>Overwrite Method</th>
<th>Overwrite Target</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATPhtml</td>
<td>buffer overflow</td>
<td>return address</td>
<td>✓</td>
</tr>
<tr>
<td>synthetic</td>
<td>buffer overflow</td>
<td>function pointer</td>
<td>✓</td>
</tr>
<tr>
<td>synthetic</td>
<td>buffer overflow</td>
<td>format string</td>
<td>✓</td>
</tr>
<tr>
<td>synthetic</td>
<td>format string</td>
<td>none (info leak)</td>
<td>✓</td>
</tr>
<tr>
<td>cfingerd</td>
<td>syslog format string</td>
<td>GOT entry</td>
<td>✓</td>
</tr>
<tr>
<td>wu-ftp</td>
<td>vsnprintf format string</td>
<td>return address</td>
<td>✓</td>
</tr>
</tbody>
</table>
Evaluation - Evaluation of attack detection

- Synthetic exploits
  - They wrote small programs for:

<table>
<thead>
<tr>
<th>Return Address</th>
<th>Function Pointer</th>
<th>Format String</th>
</tr>
</thead>
<tbody>
<tr>
<td>“gets” for long input</td>
<td>Same</td>
<td>Line input from user</td>
</tr>
<tr>
<td>Overwrote the stack – overwrote return address</td>
<td>Overwrote the stack – overwrote function pointer</td>
<td>Overwrote format string</td>
</tr>
<tr>
<td>Attack detected as return addr was tainted from user</td>
<td>Attack detected as func pointer was tainted from user</td>
<td>TaintCheck determined correctly when the format string was tainted</td>
</tr>
</tbody>
</table>
# Evaluation - Evaluation of attack detection

- Actual exploits: TaintCheck evaluated on exploits to three vulnerable servers: a web server, a finger daemon, and an FTP server.

<table>
<thead>
<tr>
<th>ATPhtpd exploit</th>
<th>cfingerd exploit</th>
<th>wu-ftpd exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web server program</td>
<td>Finger daemon</td>
<td>ftp</td>
</tr>
<tr>
<td>Ver 0.4b and lower are vulnerable to buffer overflow</td>
<td>Ver 1.4.2 and lower are vulnerable to format string</td>
<td>Version 2.6.0 of wu-ftpd has a format string vulnerability in a call to vsnprintf.</td>
</tr>
<tr>
<td>malicious GET request with a very long file name (shellcode and a return address) was sent to server. Return address overwritten so when func returns it jumps to shell code inside the file name remote shell for attacker</td>
<td>When prompts for a user name, exploit responds with a string beginning with “version” + malicious code - cfingerd copies the whole string into memory, but only reads to the end of the string “version”. Malicious code in memory starts working</td>
<td>Format string to overwrite the return address was detected</td>
</tr>
<tr>
<td>TaintCheck detected return addr was tainted and identified the new value</td>
<td>Detected also</td>
<td>TaintCheck successfully detects both that the format string supplied to vsnprintf is tainted, and that the overwritten return address is tainted.</td>
</tr>
</tbody>
</table>
Performance Evaluation – CPU Bound Process

- Hardware: 2.00 GHz Pentium 4, 512 MB RAM, RedHat 8.0
- Application: bzip2(15mb)
  - Normal runtime 8.2s
  - Valgrind nullgrind skin runtime: 25.6s (3.1x)
  - Memcheck runtime: 109s (13.3x)
  - TaintCheck runtime: 305s (37.2x)
Automatic Signature Generation

- Automatic semantic analysis based signature generation
  - Find value used to override return address – typically fixed value in the exploit code
  - Sometimes as little as 3 bytes! See paper for details

[Papadopoulos’11]
Other Applications

- ?
Acknowledgments/References (1/2)


Acknowledgments/References (2/2)
