How to Sandbox IIS Automatically without 0 False Positive and Negative

Professor Tzi-cker Chiueh

Computer Science Department
Stony Brook University
chiueh@cs.sunysb.edu
Big Picture

Ways to get malicious code/data into victim sites

(1) Break cryptography
(2) Exploit design flaws in security protocols
(3) Leverage applications’ convenience features
(4) Exploit application-level implementation bugs
(5) Exploit language-level implementation bugs
(6) Non-technical attacks: insider, social engineering, etc.

- The majority of attacks are based on (3), (4) and (5)
Software Security

- Bugs in programs lead to vulnerabilities that attackers exploit
- Design vs. Implementation bugs
- How to detect security-related bugs
  - Static analysis
  - Dynamic checking
  - Intrusion detection/prevention
Control- Hijacking Attacks

- Network applications whose control gets hijacked because of software bugs: Most worms, including MS Blast, exploit such vulnerabilities
- Three-step recipe:
  - Insert malicious code/data into the victim application
    - Sneaking weapons into a plane
  - Trick the attacked application to transfer control to the inserted code or some existing code
    - Taking over the victim plane
  - Execute damaging system calls as the owner of the attacked application process
    - Hit a target with the hijacked plane
Control-Hijacking Attack

- Three types of overflows:
  - buffer overflow
  - integer overflow
  - input argument list overflow (format string attack)

- Consequences
  - Code Injection
  - Return-to-libc
  - Data attack
Example: Stack Overflow Attack

```c
main() {
    input();
}

input() {
    int i = 0;
    int userID[5];

    while ((scanf("%d", &(userID[i]))) != EOF)
        i ++;
}
```

STACK LAYOUT

FP  124 Return address of input()  100
    120 Local variable i
    116 userID[4]
    112 userID[3]
    108 userID[2]
    104 userID[1]

SP  100 userID[0]
Proposed Defenses

Stop the attack at either of the three steps:

- Overflowing some data structures
  
  Bounds checking compiler, e.g., CASH (world’s fastest array bound checking compiler on Linux/X86 platform)

- Triggering control transfer

  Branch target check, e.g., FOOD (Foreign code detection on Windows/X86 platform)

- Issuing damaging system calls

  System call pattern check, e.g., PAID
Program semantics-Aware Intrusion Detection (PAID)

- As a last line of defense, prevent intruders from causing damages even when they successfully take control of a target victim application
- Key observation: Most damages can only be done through system calls, including denial of service attacks
- Idea: Prevent a hijacked application from issuing system calls that deviate from its semantic model
System Call Model Checking

- Achilles Heel: How to derive a system call model for an arbitrary application?
  - Manual specification: error-prone, labor intensive, non-scalable
  - Machine learning: error-prone, training efforts required
- PAID’s approach: Use compiler to extract the *sites* and *ordering* of system calls from the source code of any given application automatically
  - Guarantees zero false positives and very-close-to-zero false negatives
  - System call policy is extracted automatically and accurately
PAID Architecture

Compile Time Extraction

Application

Compiler

System Call Graph

Run Time Checking

Application

System Call Graph

Legitimacy Check

User

Kernel
System Call Flow Graph

- Take a program’s control flow graph, and eliminate all nodes that are not related to system calls
- Traverse the SCFG at run time to verify the legitimacy of every incoming system call
- Non-determinism:
  - If-then-else statements
  - Function with multiple call sites
System Call Instance Coordinate

- Each system call instance is uniquely identified by:
  - The sequence of return addresses used in the function call chain leading to the corresponding “int 80” instruction
  - The return address of the “int 80” instruction itself

- Example:
  
  Main ➔ F1 ➔ F2 ➔ F4 ➔ system_call_1 vs.
  Main ➔ F3 ➔ F5 ➔ F4 ➔ system_call_1
System Call Flow Graph Traversal

- Is there a path from the previous system call instance \((R_1, R_2, R_3, \ldots R_n)\) to the current system call instance \((S_1, S_2, S_3, \ldots S_m)\)?
- Largely deterministic \(\Rightarrow\) low latency
Dynamic Branch Targets

- Not all branch targets are known at compile time: function pointers and indirect jumps
- Insert a `notify` system call to tell the kernel the target address of these indirect branch instructions
- The kernel moves the current cursor of the system call graph to the designated target accordingly
- Notify system call is itself protected
Asynchronous Control Transfer

- **Setjmp/Longjmp**
  - At the time of `setjmp()`, store the current cursor
  - At the time of `longjmp()`, restore the current cursor

- **Signal handler**
  - When signal is delivered, store the current cursor
  - After signal handler is done, restore the current cursor

- **Dynamically linked library** such as `dlopen()`
  - Load the library’s system call graph at run time
Mimicry Attack

- Hijack the control of a victim application by overwriting some control-sensitive data structure, such as return address
- Issue a legitimate sequence of system calls after the hijack point to fool the IDS until reaching a desired system call, e.g., exec()
- None of existing commercial host-based IDS can handle mimicry attacks
Mimicry Attack Example

- Legitimate sequence:
  \[
  \text{open()} \rightarrow \text{read()} \rightarrow \text{receive()} \rightarrow \text{send()} \rightarrow \text{exec()}
  \]

- Buffer overflow vulnerability exists between `open()` and `read()`
  - Hijack the program’s control between `open()` and `read()`
  - Execute `read() \rightarrow \text{receive()} \rightarrow \text{send()} \rightarrow \text{exec()}`
Mimicry Attack Details

- To mount a mimicry attack, attacker needs to:
  - Issue each intermediate system call without being detected
    Nearly all system calls can be turned into no-ops
    For example \texttt{(void) getpid()} or \texttt{open(NULL, 0)}
  - Grab the control back after each intermediate system call
    Set up the stack so that the injected code can take control after each system call invocation
Countermeasures

- Minimize non-determinism in the system call model
  - If \( a > 1 \) \{ open(..) \} else \{open(..); write(..)\}

- Checking system call argument values whenever possible

- Random insertion of null system calls at load time to defeat guessing
  - Different SCFGs for different instances of the same program
Impossible Path Example

main()
{
    foo();   % W
    foo();   % X
    exit();  % E
}

foo()
{
    for(....){
        sys_foo(); % Y
        sys_foo(); % Z
    }
}

1/10/06
With PAID

- Legitimate Path:
  \[ WY \rightarrow WZ \rightarrow XY \rightarrow XZ \rightarrow E \]

- Impossible Path:
  \[ WY \rightarrow WZ \rightarrow E \times \]
PAID Checks

- Ordering
- Site: return address sequence
- Arguments
- Checking performed in the kernel with SCFG stored in the user space
System Call Argument Check

- Start from each “file name” system call argument, e.g., open() and exec(), and compute a backward slice towards the “inputs”

- Perform symbolic constant propagation through the slice, and the result could be
  - A constant: static constant
  - A program segment that depends on initialization-time inputs only: dynamic constant
  - A program segment that depends on run-time inputs: dynamic variables
Dynamic Variables

- Derive partial constraints, e.g., prefix or suffix, “/home/httpd/html”
- Enforce the system call argument computation path by inserting null system calls between where dynamic inputs are entered and where the corresponding system call arguments are used
Ordering Check Only

Buffer Overflow

Compromised!
Ordering and Site Check

Buffer Overflow

Compromised!

Function call

Int 0x80 instruction

setreuid
read
open
stat
write
Ordering, Site and Stack Check (1)

- Buffer Overflow
- setreuid
- read
- open
- stat
- write

function call
int 0x80 instruction
Ordering, Site and Stack Check (2)

Buffer Overflow

Stack check passes

function call

int 0x80 instruction

main

exec
Random Insertion of Notify Calls

Buffer Overflow

Attack failed

notify

exec

function call

int 0x80 instruction
Window of Vulnerabilities

Desired system call follows
Immediately

Argument replacement
Prototype Implementation

- GCC 3.1 and Gnu ld 2.11.94, Red Hat Linux 7.2
- Compiles GLIBC successfully
- Compiles several production-mode network server applications successfully, including Apache-1.3.20, Qpopper-4.0, Sendmail-8.11.3, Wuftpdl-2.6.0, etc.
## Throughput Overhead

<table>
<thead>
<tr>
<th></th>
<th>PAID</th>
<th>PAID/stack</th>
<th>PAID/random</th>
<th>PAID/stack random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>4.89%</td>
<td>5.39%</td>
<td>6.48%</td>
<td>7.09%</td>
</tr>
<tr>
<td>Qpopper</td>
<td>5.38%</td>
<td>5.52%</td>
<td>6.03%</td>
<td>6.22%</td>
</tr>
<tr>
<td>Sendmail</td>
<td>6.81%</td>
<td>7.73%</td>
<td>9.36%</td>
<td>10.44%</td>
</tr>
<tr>
<td>WuftpD</td>
<td>2.23%</td>
<td>2.69%</td>
<td>3.60%</td>
<td>4.38%</td>
</tr>
</tbody>
</table>
However

- PAID assumes source code availability, but most users do not have access to the source code of their applications, especially on the Windows platform.
- What is the SCFG for Microsoft’s IIS?
- Enters the BIRD (Binary Interpretation using Runtime Disassembly) project.
- Binary PAID = BIRD + PAID
Motivation

- Many state-of-the-art solutions to software security problem are based on program transformation techniques.
- Achilles Heel: cannot be applied to existing executable binaries, especially for Windows PE32 binaries.
- From source code to binary code:
  - Static disassembly does not always work.
  - Binary instrumentation is non-trivial.
Static Disassembly

- No guarantee for 100% recovery: no way to know for sure
- Distinguishing between instruction and data is fundamentally undecidable
- Linear sweeping: data (e.g., jump table) could be embedded code section
- Recursive traversal: some functions do not any explicit call sites in the binary
- Windows DLLs are full of hand-crafted code sequences designed to defeat reverse engineering tools
- Bottom line: about 90% coverage with absolute confidence
BIRD

- A binary analysis and instrumentation infrastructure on the Windows platform
  - Do as much static disassembly as possible
  - Uncover “statically unknown” instructions through **dynamic** invocation of disassembler
  - Provide an API for developers to add application-specific analysis and/or instrumentation routines
  - Guarantee 100% disassembly accuracy and coverage
Architecture

Checking Engine → Dynamic Disassembler → Instrumentation Engine

BIRD Runtime Engine

Static-patched Binary

Exe. Aux. File Info

Static Disassembler

Win32 Exe. File X
Dynamic Disassembly

- Statically redirect all indirect jumps/calls to a check() routine
- Redirect delivery of exception handlers to the check() routine also
- In the check() routine
  - Check if the target address is known or not
  - If known, jump to the target; else invoke the dynamic disassembler to disassemble the target area and jump
  - Update the unknown-area list and modify indirect jumps/calls in dynamically disassembled instructions
Binary Instrumentation

- Need to find enough bytes in a given instrumentation point to put in a 5-byte jump instruction
- Can use neighboring instructions only if they are not targets of other direct jump instructions in the same function
- Use INT 3 as a fall-back mechanism, which goes through an exception handler to invoke check()
Performance Penalty

- Works for all programs in MS Office suite and IE
- Latency overhead

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Original</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>Encrypt a 10MB file</td>
<td>3.4%</td>
<td>0.18%</td>
</tr>
<tr>
<td>comp</td>
<td>Compare two similar 5MB files</td>
<td>10.0%</td>
<td>0.15%</td>
</tr>
<tr>
<td>strings</td>
<td>List all strings in a binary file</td>
<td>6.4%</td>
<td>2.4%</td>
</tr>
<tr>
<td>find</td>
<td>Find a string in a 500KB file</td>
<td>19.0%</td>
<td>16.7%</td>
</tr>
<tr>
<td>objdump</td>
<td>Show object headers in an EXE file</td>
<td>3.5%</td>
<td>0.8%</td>
</tr>
</tbody>
</table>
Binary PAID

1. Win32 Executable File
2. Disassembler
3. Binary Analysis & Instrumentation
4. CSFG Generator
5. Executable Image
6. Dyncheck.dll:
   - disassembler, analysis, instrumentation, CSFG gen.
7. Sandboxing Engine

Execution
Data Flow
Control Flow

Static Time
Run Time
### Throughput Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>BIRD</th>
<th>BIRD+ BASS</th>
<th>BIRD+BASS +Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>99.9%</td>
<td>94.2%</td>
<td>94.0%</td>
</tr>
<tr>
<td>BIND</td>
<td>97.8%</td>
<td>92.3%</td>
<td>91.9%</td>
</tr>
<tr>
<td>IIS W3 Service</td>
<td>99.1%</td>
<td>93.9%</td>
<td>93.5%</td>
</tr>
<tr>
<td>MTSEmail</td>
<td>99.7%</td>
<td>97.3%</td>
<td>97.3%</td>
</tr>
<tr>
<td>Cerberus Ftpd</td>
<td>99.2%</td>
<td>93.0%</td>
<td>93.0%</td>
</tr>
<tr>
<td>GuildFTPd</td>
<td>79.9%</td>
<td>73.3%</td>
<td>71.3%</td>
</tr>
<tr>
<td>BFTelnetd</td>
<td>99.9%</td>
<td>97.4%</td>
<td>96.9%</td>
</tr>
</tbody>
</table>
Other Application: FOOD

- Goal: Ensure no dynamically injected code can run by monitoring target addresses of all indirect branches
- Assumption: no self modifying code, thus read-only text segment
- Approach: check the legitimacy of each instruction based on its **location** rather than its **content**
- Intercept at all indirect jumps/calls, return instructions and invocation of exception handlers
- Overhead: 10-25%
Conclusion

- PAID is the most efficient, comprehensive and accurate host-based intrusion prevention (HIPS) system on both Linux and Windows platform
- **Automatically** generates per-application system call policy
- Guarantee 0 false positive and **almost 0** false negative
- Effective countermeasures against mimicry attacks,
  - Extensive system call argument checks
  - Load-time insertion of random null system calls
  - Return address sequence check
- Can handle function pointers, asynchronous control transfer, threads, exceptions, and DLL
Future Work

- Further reduce the latency/throughput overhead of Binary PAID
- Reduce the percentage of “dynamic variable” category of system call arguments
- Apply it to generate security policy for SELinux automatically
- Create a counterpart of PAID for NIDS
For more information

Project Page: http://www.ecsl.cs.sunysb.edu/PAID

Thank You!