Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense

Jinpeng Wei, Bryan D. Payne, Jonathon Giffin, Calton Pu

Georgia Institute of Technology

Annual Computer Security Applications Conference (ACSAC 2008)
The Botnet Threat

- Botnet: a collection of compromised computers under the control of a malicious server or master.
- Malware (e.g., rootkits) on each bot has become increasingly sophisticated and stealthy to evade detection and removal.
- We are mainly interested in the stealthy hiding of malware in the kernel space.
Outline

• Soft timers and soft-timer-driven attacks
• Design of the STIR defense
• Implementation and evaluation
• Related work
• Conclusion
Outline

• Soft timers and soft-timer-driven attacks
  • Design of the STIR defense
  • Implementation and evaluation
  • Related work
  • Conclusion
Classification of Stealthy Control Flow Attacks in the Kernel

- Detour attacks
- Persistent control flow attacks (hooks)
- Transient control flow attacks
  - Soft-timer-driven attacks
The Soft-timer Queue

- A dynamic schedulable queue in the kernel
- Can be used to inject transient control flows

![Diagram showing the Soft-timer Queue structure and the `add_timer` function]
Soft-timer-driven Control Flow Attacks

1. schedule
2. wait
3. callback
4. run

function data expires

Legitimate Driver

function data expires

Legitimate Driver

function data expires

Legitimate Driver

Soft Timer Queue Engine

timer->function (timer->data) {
  ...
}

...
Soft-timer-driven Control Flow Attacks

1. schedule

2. wait

3. callback

4. run

Malware Module

Legitimate Driver

Legitimate Driver

function data expires

function data expires

function data expires

Soft Timer Queue Engine

timer->function (timer->data) {
  ...
}

...
Proof of Concept Malware

• How do they work?
  – Request the first STIR to interpose on the kernel control flow at break-in
  – Execute when the first STIR expires (a callback)
  – Before giving up control, request the next STIR
  – Wait for the next callback to happen

• What can they do?
  – Collect confidential information (stealthy key logger)
  – Mount a DoS attack (stealthy cycle stealer)
  – Schedule a hidden process (alter-scheduler)

STIR: Soft-Timer Interrupt Request
The Stealthy Key Logger

- Runs in Linux kernel 2.6.16
- Periodically reads the TTY line discipline buffer in the kernel, which can keep a history of up to 2,048 keystrokes
- Timer period is 1 second
Outline

• Soft timers and soft-timer-driven attacks
• Design of the STIR defense
• Implementation and evaluation
• Related work
• Conclusion
Defending Soft-timer-driven Attacks

• Main idea: a soft-timer callback function and its callees (functions it calls) should always target the trusted code of the kernel during the execution of the callback function.

• By preserving such invariants, we can defeat soft-timer-driven attacks.

Q: How can we draw the line between trusted and untrusted parts of the kernel?

A: Validate the targets of indirect control transfers. E.g., what can timer→function legally point to?
Basic Defense Strategy

• Check the callback function against a white list of legitimate callback functions
• Check the callback function as well as the data pointer
• Smarter malware may supply a legitimate callback function but a malicious data pointer (similar to the “jump-to-libc” style attacks).
High-Level View of the Defense

Input: function, data
Output: yes/no

function = ?

ab_cleanup  dev_watchdog ... tcp_delack_timer  zf_ping

data -> tx_timeout = ? ... 

ace_watchdog ... e1000_tx_timeout

How do we build and use this whitelist tree?
STIR Summary Signatures

- summary_signature := <function, assertion>
- assertion := true | dpred AND assertion
- dpred := deref equals (functionlist)
- functionlist := function | function OR functionlist

Example summary signature:

< dev_watchdog, data->tx_timeout equals (e1000_tx_timeout OR xircom_tx_timeout) >
Processing STIR Summary Signatures

Linux Kernel Source

STIR Analyzer
Symbolic STIR Signatures

Compile Time

Runtime symbol information

STIR Symbol Mapper

Guest VM

STIR Symbol Resolver

Resolved STIR Signature Database

STIR Checker

Security VM

STIR Dispatcher

Initialization Time

Evaluation Time

Evaluated STIR Signatures
Static Analysis Overview

- Top-Level Analysis
- Transitive Closure Analysis

< function, assertion >

STIR summary signature
Top-Level Analysis

/* Linux kernel 2.6.16/net/sched/sch_generic.c */

static void dev_watchdog_init(struct net_device *dev)
{
    init_timer(&dev->watchdog_timer);
    dev->watchdog_timer.data = (unsigned long)dev;
    dev->watchdog_timer.function = dev_watchdog;
}

• Traverse each assignment statement (lval = rval) in the kernel, if lval ends with a field named function within a structure of type timer_list, then rval is recognized as a soft timer callback function
Top-Level Analysis

/* Linux kernel 2.6.16/net/sched/sch_generic.c */

static void dev_watchdog_init(struct net_device *dev)
{
    init_timer(&dev->watchdog_timer);
    dev->watchdog_timer.data = (unsigned long)dev;
    dev->watchdog_timer.function = dev_watchdog;
}

- Traverse each assignment statement (lval = rval) in the kernel, if lval ends with a field named function within a structure of type timer_list, then rval is recognized as a soft timer callback function
Top-Level Analysis

/* Linux kernel 2.6.16/net/sched/sch_generic.c */

static void dev_watchdog_init(struct net_device *dev)
{
    init_timer(&dev->watchdog_timer);
    dev->watchdog_timer.data = (unsigned long)dev;
    dev->watchdog_timer.function = dev_watchdog;
}

• Traverse each assignment statement (lval = rval) in the kernel, if lval ends with a field named function within a structure of type timer_list, then rval is recognized as a soft timer callback function
Transitive Closure Analysis

static void
dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct
        net_device *)arg;
    if (dev->qdisc != &noop_qdisc) {
        ...
        printk(KERN_INFO "...%s...
", dev->name);
        dev->tx_timeout(dev);
        ...
    }

- Objective: To identify the constraints on the “data” attribute of a legitimate STIR
static void
dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct
    net_device *)arg;
if (dev->qdisc != &noop_qdisc) {
    ...
    printk(KERN_INFO "...%s...\n",
    dev->name);
    dev->tx_timeout(dev);
    ...
}
static void
dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct
    net_device *)arg;
    if (dev->qdisc != &noop_qdisc) {
        
        printk(KERN_INFO "...%s...
",  
        dev->name);
        dev->tx_timeout(dev);
        
    }
}
Transitive Closure Analysis

tainted_vars:

static void
dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct
    net_device *)arg;
    if (dev->qdisc != &noop_qdisc) {
        {arg, dev}
        ...
        printk(KERN_INFO "...%s...\n",
                dev->name);
        dev->tx_timeout(dev);
        ...
    }
}
static void
dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct net_device *)arg;
    if (dev->qdisc != &noop_qdisc) {
        printk(KERN_INFO "...%s...
", dev->name);
        dev->tx_timeout(dev);
    }
}

tainted_vars:
{arg, dev}
Transitive Closure Analysis

tainted_vars:

static void dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct net_device *)arg;
    if (dev->qdisc != &noop_qdisc) {
        printk(KERN_INFO "...%s...\n", dev->name);
        dev->tx_timeout(dev);
        ...
    }
}

{arg, dev}

Question 1:

dev->tx_timeout = ?

Answer:

Find all legitimate functions that can be assigned to the “tx_timeout” field of a structure of type “net_device”

⇒ top-level analysis
Transitive Closure Analysis

static void
dev_watchdog(unsigned long arg)
{
    struct net_device *dev = (struct net_device *)arg;
    if (dev->qdisc != &noop_qdisc) {
        ...
        printk(KERN_INFO "...%s...
", dev->name);
        dev->tx_timeout(dev);
    ...
}

Question 2:
How is the control flow of dev->tx_timeout influenced by dev?

Answer:
Perform a transitive closure analysis on the target function.

static void ariadne_tx_timeout(struct net_device *dev)
{
    volatile struct Am79C960 *lance = (struct Am79C960*)dev->base_addr;
    ...
}
Processing STIR Summary Signatures

Linux Kernel Source

STIR Analyzer

STIR Symbol Mapper

STIR Symbol Resolver

Resolved STIR Signature Database

STIR Dispatcher

Guest VM

Security VM

Compile Time

Initialization Time

Evaluation Time

Runtime symbol information

dev_watchdog, c02ae890 neigh_table_clear, c02a4810

……
Checking STIRs

Diagram:

- Linux Kernel Source
- STIR Analyzer
- Symbolic STIR Signatures
  - STIR Symbol Mapper
  - STIR Symbol Resolver
  - Resolved STIR Signature Database
  - STIR Dispatcher
  - STIR Checker

Time Phases:
- Compile Time
- Initialization Time
- Evaluation Time
STIR Checking Architecture

Guest VM

STIR Dispatcher

yes / no (function, data)

Security VM

Resolved STIR Sig. DB

STIR Checker

VMI

Xen

function = ?

ab_cleanup dev_watchdog... tcp_delack_timer zf_ping
data → tx_timeout = ?

ace_watchdog... e1000_tx_timeout...
Outline

• Soft timers and soft-timer-driven attacks
• Design of the STIR defense
• Implementation and evaluation
• Related work
• Conclusion
Implementation of the STIR Analyzer

• Based on CIL (C Intermediate Language)
• Comprised of several analysis modules
  – A top-level analyzer
  – A transitive closure analyzer
  – A type analyzer
  – Shell scripts to compose the modules
Implementation of the STIR Checking

• On top of Xen 3.0.4
• Used the VT (virtualization technology) support of an Intel CPU
• Based on the Lares architecture
Evaluation: Security Assumptions

• The VMM and the Security VM are part of the TCB (Trusted Computing Base).
• The legitimate kernel code in the guest VM’s memory can not be tampered with.
• The source code of the kernel and all kernel extensions are available for static analysis.
• The guest system can be booted into a known good state (e.g., secure boot).
Evaluation: Static Analysis Results

- We found 365 top-level callback functions in 3,688 kernel source files analyzed.
- The majority of these STIR callback functions do not derive function pointers from the input parameter.
- 32 of them need transitive closure analysis.
Evaluation: Effectiveness of Defense

- Attack experiments: can detect the sample malware.
- Can have no false negatives because it mediates every STIR execution and prevents the execution of all unknown, illegitimate STIRs.
- Can have no false positives because all potential legitimate STIRs are captured in the summary signature database.
### Evaluation: Execution Time Overhead

<table>
<thead>
<tr>
<th></th>
<th>cat</th>
<th>ccrypt</th>
<th>gzip</th>
<th>cp</th>
<th>make</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original</strong></td>
<td>20.85</td>
<td>3.30</td>
<td>5.92</td>
<td>43.95</td>
<td>217.95</td>
</tr>
<tr>
<td>(seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>STIR-aware</strong></td>
<td>20.96</td>
<td>3.30</td>
<td>6.01</td>
<td>46.61</td>
<td>218.58</td>
</tr>
<tr>
<td>(seconds)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>0.52%</td>
<td>0%</td>
<td>1.52%</td>
<td>6.05%</td>
<td>0.29%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Callbacks/Sec</strong></td>
<td>46.9</td>
<td>46.3</td>
<td>47.3</td>
<td>61.4</td>
<td>81.6</td>
</tr>
</tbody>
</table>

- **cat** - read and display the content of 8,000 small files (with size ranging from 5K to 7.5K bytes).
- **ccrypt** - encrypt a text stream of 64M bytes.
- **gzip** - compress a text file of 64M bytes using the --best option.
- **cp** - recursively copy a Linux kernel source tree.
- **make** - perform a full build of the Apache HTTP server (version 2.2.2) from source.
Evaluation: Network Throughput Overhead

- We used the Iperf-2.0.2 benchmark.
- The security VM ran the Iperf server and the guest VM ran the Iperf client.
- The experiment was run for 60 seconds, using 64KB buffers and 10 concurrent connections.

Performance drop: 4.1%
Callback frequency: 287/second
Outline

• Soft timers and soft-timer-driven attacks
• Design of the STIR defense
• Implementation and evaluation
• Related work
• Conclusion
Related Work

• Focused on code changes
  – Tripwire (a file system integrity checker)
  – IMA (TCG-based, load-time kernel and application integrity checker)
  – Copilot (coprocessor-based, run-time kernel integrity checker)
  – Pioneer (purely software-based run-time integrity checker)

• Focused on data changes
  – SBCFI (state-based control flow integrity), a sampling-based checker targeting persistent kernel control flow attacks
  – CFI (control flow integrity), checking the dynamic execution flow of a program against a statically computed control flow graph
Outline

- Soft timers and soft-timer-driven attacks
- Design of the STIR defense
- Implementation and evaluation
- Related work
- Conclusion
Conclusions

• The Soft-timer mechanism of a modern kernel provides a novel hiding technique for the malware.
• We develop a white list approach for defending against such malware.
• We use static analysis to derive the white list.
• We use virtualization to implement the defense.