Temporal System Call Specialization for Attack Surface Reduction

Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, Michalis Polychronakis
{sghavamnia, tpalit, shmishra, mikepo}@cs.stonybrook.edu
• Applications typically include code they don’t use and have access to features they don’t need
  • Some modules/plugins are not needed by a given configuration
  • Some library functions are not imported at all
  • Some system calls are never used

#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char** argv){
    char dest[1024];
    if ( argc == 2 ){
        strcpy(dest, argv[1]);
        int fd = open(dest,O_CREAT);
    }
    return 0;
}
Software Debloating and Specialization

• This “code bloat” has security implications
  • Unneeded code: more ROP gadgets for writing code reuse exploits
  • Unused (dangerous) system calls: exploit code can still invoke them to perform harmful operations (e.g., `execve()`)
  • Unused system calls: entry points for exploiting kernel vulnerabilities that can lead to privilege escalation

• Our focus: reduce the attack surface by disabling system calls
  • Break exploit payloads (shellcode, ROP)
  • Neutralize kernel vulnerabilities associated with certain system calls
Existing Work: Library Debloating

• Applications typically use only a fraction of library functions

• Library debloating: remove non-imported functions from memory
  [Mulliner and Neugschwandtner ‘15] [Quach et al. ‘18] [Agadakos et al. ‘19] [Porter et al. ‘20]
  • Generate the call graph of imported shared libraries
  • Identify library function dependencies

• Caveat: the entire lifetime of the program is considered
  • If a function/system call is used even only once, it cannot be disabled

Can we disable more system calls by differentiating between a process’ different phases of execution?
Motivation

• Server applications typically perform *initialization* operations at the beginning of their execution
  • Read configuration files
  • Fork worker processes
  • Execute other programs
  • Create files and set their permissions
• Afterwards, they enter their main *serving* phase
  • Handle client requests
  • Establish connections
  ...

Temporal System Call Specialization for Attack Surface Reduction
Example: Apache Web Server

Temporal System Call Specialization for Attack Surface Reduction
Temporal System Call Specialization

• Disable additional system calls that are needed only during the *initialization* phase, after entering the *serving* phase

Examples
• Apache Httpd and Nginx invoke `execve` *only* during initialization
• Lighttpd, Bind, and Redis invoke `chmod` *only* during initialization

Disables 51% more security-critical system calls, breaking 218 more shellcodes and ROP payloads

Mitigates 13 more Linux kernel CVEs
Outline

• Introduction

• Generating the call graph
  • Pruning based on argument types
  • Pruning based on taken addresses

• Identifying the required system calls for each phase

• Enforcing system call filters after the initialization phase

• Experimental evaluation

• Conclusion
System Overview

Programmer-provided Function List

LLVM IR

SVF Andersen’s Analysis

Type-based Pruning

Address-taken Based Pruning

Precise Call Graph

Call Graph with Type-based Pruning

Imprecise Call Graph

Temporal System Call Specialization for Attack Surface Reduction

`filter(SYS_execve)`

`filter(SYS_setuid)`

`filter(SYS_setsid)`

`filter(SYS_bind)`

`filter(SYS_listen)`
Call Graph Generation

• A complete and sound call graph is required to identify unnecessary system calls
  • The use of function pointers necessitates points-to analysis
  • While sound, points-to analysis comes with severe over-approximation
• Over-approximation prevents the precise differentiation of the system call requirements between the two phases
  • No security benefit if both initialization and serving phases use the same set of system calls
• Goal: improve precision without losing soundness
Pruning based on Argument Types

• Match arguments passed to callsite with function argument types
• Consider only struct types (no primitives, no void*)
• Consider only non-variadic functions

<table>
<thead>
<tr>
<th>Callsite</th>
<th>Target Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*gic)(int r, void *d, apr_wait_t s)</td>
<td>piped_log_mnt (int p, void *m, apr_wait_t a)</td>
</tr>
<tr>
<td></td>
<td>event_run (apr_pool * p, apr_pool * pl, serv_rec * s)</td>
</tr>
</tbody>
</table>
Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only struct types (no primitives, no void*)
- Consider only non-variadic functions

**Callsite**

(*gic)(int r, void *d, apr_wait_t s)

**Target Functions**

- piped_log_mnt (int p, void *m, apr_wait_t a)
- event_run (apr_pool * p, apr_pool * pl, serv_rec * s)
Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only struct types (no primitives, no void*)
- Consider only non-variadic functions

<table>
<thead>
<tr>
<th>Callsite</th>
<th>Target Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*gic)(int r, void *d, apr_wait_t s)</td>
<td>piped_log_mnt (int p, void *m, apr_wait_t a)</td>
</tr>
<tr>
<td></td>
<td>event_run (apr_pool * p, apr_pool * pl, serv_rec * s)</td>
</tr>
</tbody>
</table>
Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only `struct` types (no primitives, no `void`*)
- Consider only non-variadic functions
Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only struct types (no primitives, no void*)
- Consider only non-variadic functions

<table>
<thead>
<tr>
<th>Callsite</th>
<th>Target Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*gic)(int r, void *d, apr_wait_t s)</td>
<td>piped_log_mnt (int p, void *m, apr_wait_t a)</td>
</tr>
<tr>
<td></td>
<td>event_run (apr_pool * p, apr_pool * pl, serv_rec * s)</td>
</tr>
</tbody>
</table>
Pruning based on Taken Addresses

- Identify where a function address is being taken (global and local)
- Check if those locations (local) are accessible from `main()`
- Prune edges to functions that are:
  - Not accessed directly \textit{and}
  - The location where the address is being taken is not accessible
- Example: address of `piped_log_mnt` is only taken in `start_module`
  - `start_module` is not accessible from `main`

<table>
<thead>
<tr>
<th>Callsite</th>
<th>Target Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(*gic)(int r, void *d, apr_wait_t s)</code></td>
<td><code>piped_log_mnt (int p, void *m, apr_wait_t a)</code></td>
</tr>
</tbody>
</table>
Outline

• Introduction
• Generating the call graph
  • Pruning based on argument types
  • Pruning based on taken addresses
• Identifying the required system calls for each phase
• Enforcing system call filters after the initialization phase
• Experimental evaluation
• Conclusion
System Call Mapping

• Glibc call graph generation
  • Map all exported functions to the system calls they use
• Generate call graph for all libraries
  • Leaves are either system calls or functions from other libraries
• Combine all call graphs to create a complete graph
System Call Mapping

• Glibc call graph generation
  • Map all exported functions to the system calls they use
• Generate call graph for all libraries
  • Leaves are either system calls or
  • Functions from other libraries
• Combine all call graphs to create a complete graph
Outline

• Introduction
• Generating the call graph
  • Pruning based on argument types
  • Pruning based on taken addresses
• Identifying the required system calls for each phase
  • Enforcing system call filters after the initialization phase
• Experimental evaluation
• Conclusion
System Call Filtering Enforcement

• Seccomp BPF
  • Standard Linux kernel facility for system call filtering
  • Filters installed by invoking the `seccomp` or `prctl` system call
  • In case of filter conflicts, the least privileged ones are considered

• Install more restrictive filters after entering the `serving phase`

Temporal System Call Specialization for Attack Surface Reduction
Outline

• Introduction
• Generating the call graph
  • Pruning based on argument types
  • Pruning based on taken addresses
• Identifying the required system calls for each phase
• Enforcing system call filters after the initialization phase
  • Experimental evaluation
• Conclusion
Lib. vs. Temporal Specialization: Retained System Calls

Number of system calls retained (out of 333 available) after applying library debloating and temporal specialization

<table>
<thead>
<tr>
<th>Application</th>
<th>Library Debloating</th>
<th>Temporal Specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Initialization</td>
</tr>
<tr>
<td>Nginx</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td><strong>Apache Httpd</strong></td>
<td><strong>105</strong></td>
<td>94</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>Bind</td>
<td>127</td>
<td>99</td>
</tr>
<tr>
<td>Memcached</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Redis</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>
## Security-critical System Calls Disabled

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Nginx</th>
<th>Apache Httpd</th>
<th>Lighttpd</th>
<th>Bind</th>
<th>Memcached</th>
<th>Redis</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>execveat</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>execve</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fork</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ptrace</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>chmod</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>mprotect</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>setgid</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>setreuid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>setuid</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

✓: Syscall is removed  ✗: Syscall is not removed  ❖: Can be removed by applying configuration-driven specialization  *: Can be removed by applying API specialization
• Collected 567 shellcodes and 17 ROP payloads
• Increased set of shellcodes to 1,726 by generating shellcode variations based on equivalent system calls
  • Example: accept and accept4 are equivalent

Average number (%) of payloads broken by library and temporal specialization across applications

<table>
<thead>
<tr>
<th>Payload Category</th>
<th>Count</th>
<th>Library Debloating</th>
<th>Temporal Specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shellcode</td>
<td>1726</td>
<td>1103 (63%)</td>
<td>1321 (76%)</td>
</tr>
<tr>
<td>ROP Payloads</td>
<td>17</td>
<td>9 (52%)</td>
<td>11.6 (68%)</td>
</tr>
</tbody>
</table>
## Neutralized Linux Kernel CVEs

Kernel CVEs mitigated by filtering unneeded system calls

<table>
<thead>
<tr>
<th>CVE</th>
<th>System Call(s)</th>
<th>Description</th>
<th>Library</th>
<th>Temporal</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2018-18281</td>
<td>execve(at), remap</td>
<td>Allows user to gain access to a physical page after it has been released</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2016-3672</td>
<td>execve(at)</td>
<td>Allows user to bypass ASLR by disabling stack consumption resource limits</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2015-3339</td>
<td>execve(at)</td>
<td>Race condition allows privilege escalation by executing program</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2015-1593</td>
<td>execve(at)</td>
<td>Bug in stack randomization allows attackers to bypass ASLR by predicting top of stack</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2014-9585</td>
<td>execve(at)</td>
<td>ASLR protection can be bypassed due to bug in choosing memory locations</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2013-0914</td>
<td>execve(at)</td>
<td>Allows local user to bypass ASLR by executing a crafted application</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2012-4530</td>
<td>execve(at)</td>
<td>Sensitive information from the kernel can be leaked via a crafted application</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2012-3375</td>
<td>epoll_ctl</td>
<td>Denial of service can be caused due to improper checks in epoll operations</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CVE-2011-1082</td>
<td>epoll_(ctl, pwait, wait)</td>
<td>Local user can cause denial of service due to improper checks in epoll data structures</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CVE-2010-4346</td>
<td>execve(at)</td>
<td>Allows attacker to conduct NULL pointer dereference attack via a crafted application</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2010-4243</td>
<td>uselib, execve(at)</td>
<td>Denial of service can be caused via a crafted exec system call</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2010-3858</td>
<td>execve(at)</td>
<td>Denial of service can be caused due to bug in restricting stack memory consumption</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>CVE-2008-3527</td>
<td>execve(at)</td>
<td>Allows a local user to escalate privileges or cause DoS due to improper boundary checks</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Conclusion

• Temporal specialization *removes* security-critical system calls by differentiating between the execution phases of a process
  • Proposed two novel call graph pruning techniques

• Filters 51% more *security-critical system calls* than previous library debloating techniques

• Mitigates 13 more *Linux kernel CVEs* compared to previous library debloating techniques

Source code: [https://github.com/shamedgh/temporal-specialization](https://github.com/shamedgh/temporal-specialization)