Protect the System Call, Protect (most of) the World with BASTION

Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang*, Dan Williams, Changwoo Min
Takeaway

● System Calls are important
  ○ Core API interface between processes and the Operating System
  ○ Prevalent medium for code reuse to compromise entire system from a vulnerable application

● Minimal guarding of System Calls
  ○ Linux seccomp
  ○ Eliminating surface area instead of eliminating abuse
  ○ Coarse-grained defenses

● System Call Integrity: A targeted methodology to shore up system call defenses
  ○ Protection of the system, not protection of the application
  ○ Fine-grained & specialized protection that is efficient and strong
Medium for Critical Attacks

- Many **code re-use attacks end-goal** require leveraging a system call
  - Memory vulnerabilities continue to persist
  - Attacker *intermediate* steps may cause undefined behavior in application
  - But, cannot leave application process scope **without system call**
- Majority system calls are **non-security sensitive**

![Attack surface of Linux System Calls](image-url)

- Process 1
- Process ...
- Process N
Medium for Critical Attacks

- Many **code re-use attacks end-goal** require leveraging a system call
  - Memory vulnerabilities continue to persist
  - Attacker *intermediate* steps may cause undefined behavior in application
  - But, cannot leave application process scope **without system call**
- Majority system calls are **non-security sensitive**
Medium for Critical Attacks

- Many **code re-use attacks end-goal** require **leveraging a system call**
  - Memory vulnerabilities continue to persist
  - Attacker *intermediate* steps may cause undefined behavior in application
  - But, cannot leave application process scope **without system call**
- Majority system calls are **non-security sensitive**
System Call Defenses *(and why they don’t do enough)*

**Defenses**

- **Linux seccomp**
  - *Linux deployed coarse-grained allowlist/denylist*

- **Automated System Call Filtering**
  - *sysfilter: Automated system call filtering for commodity software [RAID’20]*

- **Refined Whitelisting**
  - *Temporal System Call Specialization [USENIX Sec’20]*

**Bottom Line**

- Coarse-grained filtering is not sufficient
- System calls cannot be disabled because of *core process necessity*
  - Coincidently are *targeted for attacker abuse*
  - e.g., *execve, mmap, mprotect*

- Instead of finding system call minimal set, *find meaningful context surrounding system calls!*
execve( ctx->path, ctx->argv, ctx->envp );
Our Work: Introduction of System Call Integrity

- System Call Integrity
  - Comprised of **three contexts**
  - Based on attacker pattern insight

**Attacker Pattern Insight:**
1. How are system calls invoked?
2. How are system calls reached?
3. What is passed to system calls?

```c
execve( ctx->path, ctx->argv, ctx->envp );
```
Our Work: Introduction of System Call Integrity

- System Call Integrity
  - Comprised of **three contexts**
  - Based on attacker pattern insight

**Attacker Pattern Insight:**
1. How are system calls **invoked**?
2. How are system calls reached?
3. What is passed to system calls?

```c
execve( ctx->path, ctx->argv, ctx->envp );
```
Our Work: Introduction of System Call Integrity

- System Call Integrity
  - Comprised of three contexts
  - Based on attacker pattern insight

**Attacker Pattern Insight:**
1. How are system calls **invoked**?
2. How are system calls **reached**?
3. What is passed to system calls?

**Call-Type Context**
Is this system call allowed to be called indirectly or at all?

**Control-Flow Context**
Does the live stack trace match expected program control-flow?

```c
execve( ctx->path, ctx->argv, ctx->envp );
```
Our Work: Introduction of **System Call Integrity**

- **System Call Integrity**
  - Comprised of **three contexts**
  - Based on attacker pattern insight

**Attacker Pattern Insight:**

1. **How are system calls** invoked?
2. **How are system calls** reached?
3. **What is passed** to system calls?

```c
execve( ctx->path, ctx->argv, ctx->envp );
```
**System Call Integrity - 1 - Call Type Context**

**Guarantee:** Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

---

```c
1 void foo ( int f0 ){
2
3   int flags = MAP_ANON|MAP_SHARED;
4   bar( x1, flags );
5   ...
6 }

7 void bar ( char* b1, int b2 ){
8   int prots = PROT_READ|PROT_WRITE;
9   mmap( NULL, gshm->size, prots, b2,
10       -1, 0 );
11   ...
12 }
```

---
System Call Integrity - 1 - Call Type Context

**Guarantee**: Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

```
Example

1 void foo ( int f0 ){
2     3
3     int flags = MAP_ANON|MAP_SHARED;
4     bar( x1, flags );
5     ...  
6 }
7 void bar ( char* b1, int b2 ){
8     int prots = PROT_READ|PROT_WRITE;
9     mmap( NULL, gshm->size, prots, b2,
10         -1, 0 );
11     ...
12 }
```
Example

```c
1 void foo ( int f0 ){
2 3
3   int flags = MAP_ANON|MAP_SHARED;
4   bar( x1, flags );
5   ...
6 }
7 void bar ( char*  b1, int b2 ){
8   int prots = PROT_READ|PROT_WRITE;
9   mmap( NULL, gshm->size, prots, b2,
10      -1, 0 );
11   ...
12 }
```
System Call Integrity - 1 - Call Type Context

**Guarantee:** Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

---

**Example**

```
1 void foo ( int f0 ){
2
3  int flags = MAP_ANON|MAP_SHARED;
4  bar( x1, flags );
5  ...
6 }
7 void bar ( char* b1, int b2 ){
8  int prots = PROT_READ|PROT_WRITE;
9  mmap( NULL, gshm->size, prots, b2,
10        -1, 0 );
11  ...
12 }
```
**System Call Integrity - 1 - Call Type Context**

**Guarantee:** Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

---

**Example**

```
1 void foo ( int f0 ){
2 3
3    int flags = MAP_ANON|MAP_SHARED;
4    bar( x1, flags );
5    ...
6 }
7 void bar ( char*  b1, int b2 ){
8    int prots = PROT_READ|PROT_WRITE;
9    mmap( NULL, gshm->size, prots, b2,
10           -1, 0 );
11    ...
12 }
```
**System Call Integrity - 1 - Call Type Context**

**Guarantee**: Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

---

**Example**

```c
1 void foo ( int f0 ) { 
2    
3    int flags = MAP_ANON|MAP_SHARED;  
4    bar( x1, flags );  
5    ...  
6 }
7 void bar ( char* b1, int b2 ) { 
8    int prots = PROT_READ|PROT_WRITE; 
9    mmap( NULL, gshm->size, prots, b2, 
10             -1, 0 );  
11    ...  
12 }
```

---

**VIRGINIA TECH.**
System Call Integrity - 1 - Call Type Context

**Guarantee:** Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

### Example

```c
1 void foo ( int f0 ){
2    int flags = MAP_ANON|MAP_SHARED;
3    bar( x1, flags );
4    ...
5 }}
6 void bar ( char* b1, int b2 ){
7    int prots = PROT_READ|PROT_WRITE;
8    mmap( NULL, gshm->size, prots, b2,
9           -1, 0 );
10   ...
11 }
```
System Call Integrity - 1 - Call Type Context

**Guarantee:** Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

---

**Example**

```c
1 void foo ( int f0 ){
2     int flags = MAP_ANON|MAP_SHARED;
3     bar( x1, flags );
4     ...
5 }
6 }
7 void bar ( char*  b1, int b2 ){
8     int prots = PROT_READ|PROT_WRITE;
9     mmap( NULL, gshm->size, prots, b2, -1, 0 );
10    ...
11 }
```

---

<table>
<thead>
<tr>
<th>System Call</th>
<th>Call Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmap</td>
<td>Directly-Callable</td>
</tr>
<tr>
<td>mprotect</td>
<td>Not-Callable</td>
</tr>
</tbody>
</table>

- Sensitive system call use is **sparse** & rarely invoked **indirectly.**
Guarantee: A sensitive system call is reached and invoked only through legitimate control-flow paths during runtime.

Example

```c
void foo ( int f0 ){
    int flags = MAP_ANON|MAP_SHARED;
    bar( x1, flags );
    ...
}

void bar ( char* b1, int b2 ){
    int prots = PROT_READ|PROT_WRITE;
    mmap( NULL, gshm->size, prots, b2, -1, 0 );
    ...
}
```
**Guarantee**: A sensitive system call is reached and invoked only through legitimate control-flow paths during runtime.

```c
void foo ( int f0 ){
    int flags = MAP_ANON|MAP_SHARED;
    bar( x1, flags );
    ...
}

void bar ( char* b1, int b2 ){
    int prots = PROT_READ|PROT_WRITE;
    mmap( NULL, gshm->size, prots, b2, -1, 0 );
    ...
}
```

**Valid Control Flow**

<table>
<thead>
<tr>
<th>Function A</th>
<th>Function B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>foo</td>
</tr>
<tr>
<td>mmap</td>
<td>bar</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Call chains of sensitive system calls are usually **short**!
**Guarantee:** A sensitive system call can only use valid arguments when being invoked

- **Even if** attackers have access to memory corruption vulnerabilities

**Argument Type Coverage**

- Constants
- Global Variables
- Local Variables
- Caller Parameters

---

Call depth to set system call arguments is fairly shallow – within the same function or only a few functions away.
BASTION Overview - System Call Integrity in Practice

BASTION Compiler
- Static analysis
- Record metadata
- Sensitive variable instrumentation

BASTION Runtime Monitor
- Separate process
- Leverage context metadata
- Dynamic context checking

Operating System
Every Sensitive System Call intercepted by BASTION

User Application
BASTION Compiler - Argument Integrity Context

Procedure
- Instrumented as inline assembly
- Use variable use-def chains derived from LLVM IR
- Static and dynamic variable support

```c
1 void foo ( int f0 ){
2
3
4 int flags = MAP_ANON|MAP_SHARED;
5
6
7 bar( x1, flags );
8 ...
9 }
10 void bar ( char* b1, int b2 ){
11
12 int prots = PROT_READ|PROT_WRITE;
13
14
15
16
17
18
19
20
21
22 mmap( NULL, gshm->size, prots, b2, -1, 0);
23 ...
```
1 void foo ( int f0 ){
2
3
4
5 int flags = MAP_ANON|MAP_SHARED;
6
7
8 bar( x1, flags );
9 ...
10 } 
11 void bar ( char* b1, int b2 ){ 
12
13
14
15
16
17
18
19
20
21
22 mmap( NULL, gshm->size, PROT_READ|PROT_WRITE, b2, -1, 0 );
}...

Procedure

• Instrumented as inline assembly
• Use variable use-def chains derived from LLVM IR
• Static and dynamic variable support

Instrumentation

ctx_write_mem()

• Added at each argument write operation
BASTION Compiler - Argument Integrity Context

Procedure
- Instrumented as inline assembly
- Use variable use-def chains derived from LLVM IR
- Static and dynamic variable support

Instrumentation

ctx_write_mem()
- Added at each argument write operation

ctx_bind_mem()/ctx_bind_const()
- Stages expected values for performing runtime checking
BASTION Design - Monitor Component

Monitor Goals:

- Act as liaison between application and OS
  - Safeguard system calls from arbitrary use!
- Separate process
  - Isolates BASTION from untrusted application!
  - Attacker cannot bypass/disable BASTION hooks
- Only check contexts when system call invoked
  - Minimize interference for max performance!

Runtime Monitor Procedure

BASTION Runtime Monitor

Context Metadata

Process State Information

PC & code

Stack

Registers & Instrumentation

Checking Mechanism

mmap() invoked

mmap() completed
BASTION Prototype Implementation

- BASTION Compiler
  - LLVM 10.0.0
  - ~4K LoC

- BASTION Library API
  - ~700 LoC

- BASTION Monitor
  - ~8K LoC
  - seccomp-BPF
  - ptrace

- System
  - X86-64
  - Linux 5.19.14

**Security-Sensitive System Calls (20)**

**Arbitrary Code Execution**
- execve, execveat, fork, vfork, clone, ptrace

**Memory Permission Changes**
- mprotect, mmap, mremap, remap_file_pages

**Privilege Escalation**
- chmod, setuid, setgid, setreuid

**Networking Reconfiguration**
- socket, bind, connect, listen, accept, accept4
BASTION Evaluation

Evaluation Summary

● Performance: System-call & I/O Intensive Applications
  ■ NGINX - Most widely deployed web server
  ■ SQLite - Database Engine
  ■ vsFTPd - FTP server

● Security: 32 Attack Study: ROP payloads, real-world CVEs, & synthesized attacks

Evaluation Questions

Performance
1) What is each context’s performance impact?
2) How much overall performance overhead does BASTION impose?

Security
1) How secure is BASTION?
2) How does BASTION defend against different attack strategies?
3) How does BASTION compare to other security archetypes?
**Argument Integrity** Context is BASTION’s **most expensive** context to deploy

- BASTION **overall performance overhead** is low (<2.01%)
BASTION Performance

- **Argument Integrity** Context is BASTION’s **most expensive** context to deploy
- **BASTION overall performance overhead is low** (<2.01%)
- **Argument Integrity** Context is BASTION’s **most expensive** context to deploy
- **BASTION overall performance overhead is low** (<2.01%)
- **Argument Integrity** Context is BASTION’s most expensive context to deploy
- **BASTION overall performance overhead** is low (<2.01%)
# BASTION Security Analysis

<table>
<thead>
<tr>
<th>Attack Category</th>
<th>Call Type</th>
<th>Control Flow</th>
<th>Argument Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return-Oriented Programming (18)</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>• Stack pivot gives away ROP chain</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Direct System Call Manipulation (9)</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>• Naive attacks corrupting function pointers</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Indirect System Call Manipulation (5)</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>• Advanced attacks mimic valid program behavior</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>• All attacks attempt to corrupt arguments</td>
<td></td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

- NEWTON CPI Attack [SIGSAC'17]
- AOCR Apache Attack [NDSS'17]
- AOCR NGINX Attack 2 [NDSS'17]
- COOP [S&P’15]
- Control Jujutsu [CCS’15]
Conclusion

System Calls are an attacker gateway
● Coarse-grained filtering is not enough
● System call protection needs to be fine grained to be effective

System Call Integrity
● System Call Integrity hardens system calls by applying three specialized contexts
● Specialized coverage minimizes CPU interference while maximizing security around system calls

Looking Towards the Future
● BASTION can be a stepping stone to enable configurable system call protection
● BASTION can be expanded to add future contexts to protect against yet unknown system call threats
● BASTION can be used as starting framework to protect against other system call threats
EXTRA SLIDES
BASTION System Call Statistics

- Some system calls are called more than others (e.g., `accept4` vs `connect`)
- System calls have **sparse** callsites
- System calls **very rarely invoked indirectly**
- **Constant arguments** are common

<table>
<thead>
<tr>
<th>Application</th>
<th>NGINX (32 workers)</th>
<th>SQLite</th>
<th>vsFTPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>total # application callsites</td>
<td>7,017</td>
<td>12,253</td>
<td>4,695</td>
</tr>
<tr>
<td>total # arbitrary direct callsites</td>
<td>6,692</td>
<td>12,026</td>
<td>4,688</td>
</tr>
<tr>
<td>total # arbitrary in-direct callsites</td>
<td>325</td>
<td>227</td>
<td>7</td>
</tr>
<tr>
<td>total # sensitive callsites</td>
<td>26</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>total # sensitive system calls called indirectly</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>NGINX</th>
<th>SQLite</th>
<th>vsFTPD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctx_write_mem()</td>
<td>5,226</td>
<td>1,337</td>
<td>204</td>
</tr>
<tr>
<td>ctx_bind_mem()</td>
<td>43</td>
<td>18</td>
<td>33</td>
</tr>
<tr>
<td>ctx_bind_const()</td>
<td>18</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>Total instrumentation sites</td>
<td>5,287</td>
<td>1,368</td>
<td>246</td>
</tr>
</tbody>
</table>

Note: **highlighted** cells indicate critical system calls or high invocation rates.
Other Considerations

Attacks able to bypass BASTION?
- (subset of) **Data-only attacks**
- In practice, will be difficult to overcome BASTION constraints
  - most information can be deduced from static analysis

Deploying BASTION to real-world (2 main challenges)
- performance overhead - fine-grained defenses do constant checks to minimize deviation from correct control flow

Comparison to CFI
- Call Type + Control Flow Context are **NOT equivalent to CFI**
- Call Type is **NOT per callsite**
- Control Flow is not application wide (only covers paths that eventually lead to system calls)

Effectiveness of BASTION under arbitrary memory corruption
- info gained from static analysis significantly raises security
- attacker would need to accurately recreate a fake version of all 3 contexts
- In practice this would require MANY read/write operations to match constraints all the while STILL obeying all static constraints deduced from BASTION analysis
Other Considerations 2

Selection of “Sensitive System Calls”
- Targets system calls enabling common attacker strategies aimed at escaping the scope of the victim application and reaching the underlying system
  - arbitrary code execution
  - memory permission changes
  - privilege escalation
  - network reconfiguration
- We investigated open/write system call - this imposed significant performance overhead
  - We confirmed that overhead comes from fetching process state

Other competitors - Saffire (EuroS&P’20)
- Explore fine-grained syscall filtering (of arguments)
- BASTION is more secure as Saffire is a userspace solution (works inside scope of vulnerable application) and relies on fine-grained CFI to be in place to ensure their defense is not skipped
- BASTION is faster than Saffire since the true performance cost for them is: CFI checking + Saffire checking

Selection of benchmarks
- Did not look at compute bound benchmarks because these very rarely used security-sensitive system calls
- Further, all compute benchmarks only used sysealls for initialization of datasets and importing libraries. very very rarely during computation phase
BASTION System Call Statistics 2

- Even in the case of File system system calls, there was great contrast of call count (e.g., `open` (light use) vs `write` (heavy use) use in webserver)
- Heavy system call invocation bottlenecked BASTION at context switching (userspace/kernelspace)
- Would be resolved if BASTION was implemented directly in kernel (module)

<table>
<thead>
<tr>
<th>BASTION Configuration</th>
<th>Runtime &amp; % Overhead Added Per Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NGINX</td>
</tr>
<tr>
<td>BASTION + file system syscalls (seccomp hook only)</td>
<td>110.41 (0.15%)</td>
</tr>
<tr>
<td>BASTION + file system syscalls (fetch process state)</td>
<td>4.56 (95.88%)</td>
</tr>
<tr>
<td>BASTION + file system syscalls (full context checking)</td>
<td>3.65 (96.70%)</td>
</tr>
</tbody>
</table>