Shreds: Fine-grained Execution Units with Private Memory

Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, Long Lu

RiS3 Lab / Computer Science / Stony Brook University
Execution Units

A Process

- Traditional Execution Units
  - Processes
    - Separate address spaces
  - Threads
    - Sharing one address space
In-process Memory Abuses

• Definition:

Malicious or compromised components try to steal data or execute code of other components running in the same process.

• Two examples

- Stealing secret data
  • The Heartbleed bug

- Executing private code
  • Private APIs in iOS Apps
## Potential Mitigations of in-Process Abuse

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Why unsuitable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process-level isolation (OpenSSH, Chrome)</td>
<td>• IPC is expensive</td>
</tr>
<tr>
<td></td>
<td>• Adoption effort</td>
</tr>
<tr>
<td>Software fault isolation-like techniques</td>
<td>• Require instrumenting untrusted code</td>
</tr>
<tr>
<td>(Native Client)</td>
<td>• Ineffective on dynamic or external code</td>
</tr>
<tr>
<td>Hardware-assisted techniques (SGX, Trustzone)</td>
<td>• Overly restrictive execution environment</td>
</tr>
<tr>
<td></td>
<td>• Semantic gap</td>
</tr>
</tbody>
</table>
Introducing Shred

- **Shred**
  - Arbitrarily scoped segment of a thread execution
- **S-pool**
  - The private memory pool for each shred
- **Shred APIs & OS-level supports**
Example Use Case

- Password authentication on web server\textit{(w/o shred)}
Example Use Case cont.

- Password authentication on web server (w/ shred)
Shred APIs

- **err_t shred_enter(int pool_desc);**
  - Start a shred execution on the current thread
  - Unlock s-pool

- **err_t shred_exit();**
  - Terminate a shred execution
  - Lock down the s-pool

- **void * spool_alloc(size_t size);**
  - Allocate memory inside S-pool

- **err_t spool_free(void *ptr);**
  - Free memory inside S-pool
int http_request_parse(server *srv, connection *con) {
    ... char *cur; /* to receive password */
    + if (strncmp(cur, auth_str, auth_str_len)==0){
        + shred_enter(AUTH_PASSWD_POOL);
        + /* receive and save password */
        + data_string *ds = s_ds_init();
        + int pw_len = get_passwd_length(cur);
        + cur += auth_str_len + 1;
        + buffer_copy_string_len(ds->key, auth_str, auth_str_len);
        + buffer_copy_string_len(ds->value, cur, pw_len);
        + cur += pw_len;
        + shred_exit();
    + }
    ...}
Listing 1: lighttpd/src/request.c

Shreds: Fine-grained execution units with private memory
Code Example cont.

/* called inside a shred */
data_string *s_ds_init(void) {
  data_string *ds;
  ds = spool_alloc(sizeof(*ds));
  ... return ds;
}

/* called inside a shred */
void s_ds_free(data_string *ds) {
  ... spool_free(ds->key);
  ... return;
}

/* inside HTTP auth module */
+ shred_enter(AUTH_PASSWD_POOL);
/* ds points passwd obj in spool */
http_authorization = ds->value->ptr;
/* hash passwd and compare with local copy*/
+ s_ds_free(ds);
+ shred_exit();

S-pool allocation APIs wrapper

Listing 2: lighttpd/src/data_string.c

Listing 3: lighttpd/src/mod_auth.c
System overview

- Two major components

S-driver

S-compiler
**System Component: S-driver**

- **S-driver**
  - Entry/exit of shreds
  - S-pool (de)allocations
  - Controls the access to S-pools

- **S-pool sharing**

- **Process**

- **Thread1**
  - `shred_enter (P1);`
  - `shred_exit();`

- **Thread2**
  - `shred_enter (P2);`
  - `shred_exit();`

- **S-pool**
  - P1
  - P2

- **Mem Space**

- **S-driver**

- **S-pool Manager**

- **Security Monitor**

---

Shreds: Fine-grained execution units with private memory
How S-pool is Built

- Intel: Memory protection keys

The building block

- ARM Memory Domains
  - Domain Access Control Register

: Accessible

: Not accessible

Shreds: Fine-grained execution units with private memory
Challenges & Solutions

- ARM Memory Domain was not designed to serve our goal

1) The granularity of the accessing subject can only be checked at CPU level

   ✓ Create the notion of shred so the accessing subject can be recognized and use S-driver to manage

2) Limited Domains: Only 16 Domains are available

   ✓ Statically bind an accessible domain to each CPU

   ✓ Reuse a domain for multiple S-pools if they are accessed from the same CPU

Virtual Address Space
S-pool Managements

S-driver will,

- Lock s-pool when,
  - Shred exits
  - Context-switch Out
  - Asynchronous events: signal handling, etc

- Unlock s-pool when,
  - Shred enters
  - Context-switch in
  - Resuming from asynchronous events
Moving the Domain Adjustments Off the Critical Path

• Changing PDE is relatively cumbersome
  - Page table walking
  - TLB invalidation

• TWO knobs to control the accessibility of S-pool
  - Domain of the corresponding page table entry
  - Value of corresponding DACR entry

• Changing DACR value is much faster, only one instruction
  - \texttt{MCR p15, 0, <Rd>, c3, c0, 0 ; Write DACR}
  - Develop the \texttt{domain fault handler} to handle domain fault \textit{lazily}
    • Detecting attacks
    • Recover from legitimate domain faults
Runtime Protections

- Secure stacks
  - Each shred has a secure stack allocated from its s-pool

- System interface protection
  - ptrace()
  - /dev/mem
  - Directly read secret from file
  - etc
Developers are virtually helpless when it comes to preventing in-process abuse in their programs, due to a lack of convenient, fine-grained, and efficient protection of sensitive memory content against hostile code running in the same process, such as injected shellcode and malicious libraries. We generally refer to this class of attacks as in-process abuse. There are three major limitations. First, thread-level memory isolation is still too coarse to stop in-process abuse because exploitable or malicious code often runs in the same thread as the legitimate code that needs to access sensitive memory content. Second, adopting these solutions requires significant efforts from developers. Separating application components into different threads (as opposed to regional code patches, to deal with the added concurrency. Third, threads with private memory tend to be more performing using 5 non-trivial open source software, including OpenSSL and Lighttpd. The results show that shreds are fairly easy to use and incur low runtime overhead (4.67%).
Evaluation

- Hardware spec: Raspberry Pi 2 Model B (Quad-core Cortex-A7 Processor with 1GB RAM)

**Softwares**
- Curl
- Minizip
- OpenSSH
- OpenSSL
- Lighttpd

**Easy adoption**
- Avg. 21 SLOC change
- Avg. 32 min adoption time

**Low overhead**
- Avg. 4.67% slowdown
- Avg. 7.26% RSS(resident set size) overhead
Evaluation cont.
Conclusion

- Goal — To help developers protect sensitive code/data from in-process abuse

- To achieve the goal we propose shreds with private memory
  - Fine-grained: Flexibly scoped segments of thread executions
  - Efficient and compatible: MMU based domain check
    - No multiple page tables
    - No nested paging
    - No heavy instrumentations
    - No hardware modifications
  - Robust:
    - Prevent out-shred attacks + intra-shred vulnerabilities