Bypassing clang’s SafeStack for Fun and Profit

Enes Göktaş, Angelos Economopoulos, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, Herbert Bos
Outline

• SafeStack

• Neglected Pointers

• Thread Spraying

• Allocation Oracles

• Conclusion
SafeStack

- New security feature in LLVM
- Protect against stack based control-flow hijacks

- In research proposals:
  - Code-Pointer Integrity (Kuznetsov et al., 2014) (origin SafeStack)
  - ASLR-Guard (Lu et al., 2015)

- Also proposed for integrating in GCC
Original stack
Original stack

Safe stack

Unsafe stack
What is it good against?

Original stack

Safe stack

Unsafe stack
What is it good against?

Original stack

Safe stack

Unsafe stack
What is it good against?

Original stack

Safe stack

Unsafe stack
What is it
good against?

Original stack

Safe stack

Unsafe stack
What is it good against?

Original stack

Safe stack

Unsafe stack
What is it good against?

Stack buffer overflows

Original stack

Safe stack

Unsafe stack

Stack buffer overflows
What is it good against?

Stack buffer overflows

Original stack

Safe stack

Unsafe stack
What is it good against?

Stack buffer overflows

Info. disclosure => stack loc.

Original stack

HEAP

BUF_A_ptr

What is it good against?

Safe stack

Unsafe stack

BUF_A

VAR

RET

VAR

RET

VAR

RET

VAR

RET

VAR

RET
What is it good against?

INFO. disclosure => stack loc.

Stack buffer overflows

Original stack

Safe stack

Unsafe stack
What is it good against?

- Stack buffer overflows
- Leaking stack location

Info. disclosure => stack loc.

Original stack

Safe stack

Unsafe stack
PIE compiled program in Linux

**Normal**

- High addr.
  - Stack
  - Heap
  - Data
  - Code
  - `mmap`

- Low addr.

**Compiled with SafeStack**

- High addr.
  - Safe Stack
  - Heap
  - Data
  - Code
  - `mmap`
  - Unsafe Stack

- Low addr.
PIE compiled program in Linux

Normal

Compiled with SafeStack
PIE compiled program in Linux

Normal

Compiled with SafeStack

High addr.

Stack

Heap

Data

Code

mmap

Low addr.

High addr.

Safe Stack

Heap

Data

Code

mmap

Unsafe Stack

Low addr.
Pointers to Safe Stack may not appear in reachable memory to keep Safe Stack hidden.
int main(int argc, char *argv[]){
    char buf[32];
    strcpy(buf, argv[1]);
    ...
}

Allocate address taken local variable on stack
int main(int argc, char *argv[]){
    char buf[32];
    strcpy(buf, argv[1]);
    ...
}

Allocate address taken local variable on stack

Address of variable provided to strcpy

Allocate address taken local variable on stack
SafeStack

• Compile time instrumentation pass
  • Flag: -fsanitize=safe-stack
• Ensure stack access is “safe”
  • Address taken objects moved to alternative stack
• Prevent leaking stack location
• Relies on ASLR
SafeStack

- Compile time instrumentation pass
  - Flag: -fsanitize=safe-stack
- Ensure stack access is “safe”
  - Address taken objects moved to alternative stack
- Prevent leaking stack location
- Relies on ASLR

How safe is the SafeStack?
SafeStack

• Compile time instrumentation pass
  • Flag: -fsanitize=safe-stack
• Ensure stack access is “safe”
  • Address taken objects moved to alternative stack
• Prevent leaking stack location
• Relies on ASLR

How safe is the SafeStack?
Locating SafeStack

• Neglected pointers

• Thread Spraying

• Allocation Oracles
Threat Model

- Memory corruption
- Arbitrary read/write primitive
- Heap and module data disclosed
- Goal: Locate SafeStack
Neglected Pointers

• SafeStack ensures **pointer to data on stack** wont be stored outside the stack

• Analyze programs compiled with SafeStack for unexpected pointers
  • GDB + python
  • Report pointers common among apps
Neglected Pointers

• Found pointers:
  • In heap
  • In libraries
  • Thread IDs
Neglected Pointers: Heap

- Dynamic Thread Vector (DTV)
  - Points to Thread Local Storage (TLS) blocks
  - Static TLS blocks attached to TCB
  - TCB of secondary stacks located on stack

https://www.uclibc.org/docs/tls.pdf
Neglected Pointers: Libraries

• pthread.so (linked lists):
  • stack_used – __stack_user

• libc.so
  • program_invocation_name
  • program_invocation_short_name

• libgcc.so
  • __libc_argv – __dlfcn_argv
Neglected Pointers: Libraries

- ld.so
  - rtld_global_ro
  - environ
  - _dl_argv
  - __libc_stack_end

- Pointer that can lead to TCB in ld.so
  - alloc_end
    - If app overloads malloc, e.g. Chrome and Firefox
Neglected Pointers: Thread IDs

- Surprisingly thread API uses **base of TCB** as thread IDs
  - `int pthread_create(pthread_t *thr, ..)`
  - `int pthread_join(pthread_t thr, ..)`
  - `pthread_t pthread_self()`
  - ...

- **Apps** that do thread bookkeeping store thread IDs in the **heap** or **modules** in their **data** section

- E.g. libxml2.so:
  - `.bss: mainthread = pthread_self()`
• Let’s assume these implementation issues are **fixed**

• The attacker **cannot leak** safestack through pointers anymore

• The attacker could try to **randomly hit** safestack

• What could he do to increase the chance to hit a safestack?
• Let’s assume these implementation issues are **fixed**

• The attacker **cannot leak** safestack through pointers anymore

• The attacker could try to **randomly hit** safestack

• What could he do to increase the chance to hit a safestack?

  Reduce the entropy through *Thread Spraying*
Entropy

• Degree of randomness
• Given in bits

• Example:
  • 3 bit address space
  • 8 blocks of 1 byte

• Hide data

(2^1)

000
001
010
011
100
101
110
111

Entropy: 2 bits

Hit chance: \( \frac{1}{2^2} = \frac{1}{4} \)

Worst case: #probes \( 2^2 = 4 \)
Entropy

• Degree of randomness
• Given in bits

• Example:
  • 3 bit address space
  • 8 blocks of 1 byte

• Hide data

<table>
<thead>
<tr>
<th>Entropy:</th>
<th>2 bits</th>
<th>1 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit chance:</td>
<td>$\frac{1}{2^2} = \frac{1}{4}$</td>
<td>$\frac{1}{2^1} = \frac{1}{2}$</td>
</tr>
<tr>
<td>Worst case:</td>
<td>$2^2 = 4$</td>
<td>$2^1 = 2$</td>
</tr>
<tr>
<td>#probes:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
64 bit address space
64 bit address space

Linux user space only uses 47 bit
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = 2^{12} bytes
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = 2¹² bytes

Safe Stack of 8 MB = 2²³ bytes = 2¹¹ pages

Hide: 2²³ bytes

Entropy: 2⁴ bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**
Legitimately spawn as many threads as possible
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

Thread Spraying
Legitimately spawn as many threads as possible

Spawn a new thread

Entropy: 23 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**
Legitimately spawn as many threads as possible

Spawn a new thread

Spawn 2 more threads

**Entropy:** 22 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

**Thread Spraying**
Legitimately spawn as many threads as possible

Spawn a new thread

Spawn 2 more threads

Spawn 128k threads = $2^{17}$ stacks

Entropy: 7 bits
64 bit address space

Linux user space only uses 47 bit

1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

Thread Spraying
Legitimately spawn as many threads as possible

Spawn a new thread

Spawn 2 more threads

Spawn 128k threads = $2^{17}$ stacks

Drops worst case #probes to 128

Entropy: 7 bits

Hide: $2^{40}$ bytes
64 bit address space

Linux user space only uses 47 bit page:
1 page: 4096 bytes = $2^{12}$ bytes

Safe Stack of 8 MB = $2^{23}$ bytes = $2^{11}$ pages

Thread Spraying
Legitimately spawn as many threads as possible

Spawn a new thread
Spawn 2 more threads
Spawn 128k threads = $2^{17}$ stacks

Mmap entropy is 40 bit => worst case #probes is 1 ($2^0$)

Drops worst case #probes to 128

Entropy: 7 bits
Inspected apps

- Firefox

- MySQL
Thread Spraying: Firefox

• New thread per dedicated web worker in JS
• 20 web workers per domain
• Web worker thread stack size = 2MB ; entropy = 19 bits
• 20 Threads drops entropy to about 15 bits

Linux stack entropy = 40 bits
2MB occupies 21 bits in AS
40 - 21 bits = 19 bits of entropy
#probes = 524288

#probes = 32768
Thread Spraying: Firefox

• New thread per dedicated web worker in JS
• 20 web workers per domain
• Web worker thread stack size = 2MB ; entropy = 19 bits
• 20 Threads drops entropy to about 15 bits

• Load pages from different domains through iframes
  • => Unlimited web worker threads
• 16.384 Web workers drop entropy to 5 bits
Thread Spraying: MySQL

- New thread per network connection
- Max connections 151
- Thread stack size = 256KB ; entropy = 22 bits
- 151 connections drops entropy to about 15 bits
Thread Spraying: MySQL

• New thread per network connection
• Max connections 151
• Thread stack size = 256KB ; entropy = 22 bits
• 151 connections drops entropy to about 15 bits

• 4096 connections drops entropy to 10 bits
  • max_connections = 4096
• Stack size of 256 MB can drop entropy to 0 bits
  • connection_attrib.stack_size = 0x10000000
Thread Spraying: MySQL

• New thread per network connection
• Max connections 151
• Thread stack size = 256KB ; entropy
• 151 connections drops entropy to about
• 4096 connections drops entropy to 10 bits
  • max_connections = 4096
• Stack size of 256 MB can drop entropy to 0 bits
  • connection_attrib.stack_size = 0x10000000

Exhausted 0x7F.. address region. Address 0x7F0000000000 has safestack with a very high chance.
• By spraying lots of threads
  • ASLR can be weakened
  • Chance to hit safestack can be increased

• Spraying might not always be possible

• Another approach to find the safestack:
  • Allocation Oracles
Safe Stack
↓
Heap
↓
Data
↓
Code
↓
mmap
↓
Unsafe Stack

High addr.

Low addr.
Safe Stack

Heap

Data

Code

\textit{mmap}

Unsafe Stack

\textbf{Size ??}

High addr.

Low addr.

\textbf{Holes}

56
### Size Distributions

<table>
<thead>
<tr>
<th>Hole</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>130TB</td>
<td>131TB</td>
</tr>
<tr>
<td>B</td>
<td>1GB</td>
<td>1TB</td>
</tr>
<tr>
<td>C</td>
<td>4KB</td>
<td>4GB</td>
</tr>
</tbody>
</table>

- **Safe Stack**
- **Heap**
- **Data**
- **Code**
- **mmap**
- **Unsafe Stack**

**Holes**

- **A**
- **B**
- **C**
So look for the holes

• Intuition:
  • repeatedly allocate large chunks of memory of size $L$ until we find the “right size”

Succeeds!
Sizeof(Hole) ≥ $L$
So look for the holes

- Intuition:
  - repeatedly allocate large chunks of memory of size $L$ until we find the “right size”

Too large, alloc fails!
Sizeof(Hole) < $L$
So look for the holes

- Intuition:
  - repeatedly allocate large chunks of memory of size $L$ until we find the “right size”

Succeeds!  
Sizeof(Hole) $\geq L$
So look for the holes

• Intuition:
  • repeatedly allocate large chunks of memory of size $L$ until we find the “right size”

Too large, alloc fails!
Sizeof(Hole) < L
So look for the holes

• Intuition:
  • repeatedly allocate large chunks of memory of size $L$ until we find the “right size”

Nailed it!

Binary search
Ephemeral Allocation Primitive (EAP)

- For each probe (i.e., server request):
  
  \[
  \text{ptr} = \text{malloc} (\text{size});
  \]
  
  \[
  \ldots
  \]
  
  \[
  \text{free}(\text{ptr});
  \]
  
  \[
  \text{reply} (\text{result});
  \]

- Strategy: allocation + deallocation, repeat
### Size Distributions

<table>
<thead>
<tr>
<th>Hole</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>130TB</td>
<td>131TB</td>
</tr>
<tr>
<td>B</td>
<td>1GB</td>
<td>1TB</td>
</tr>
<tr>
<td>C</td>
<td>4KB</td>
<td>4GB</td>
</tr>
</tbody>
</table>

![Diagram showing memory regions and size distributions](image-url)
<table>
<thead>
<tr>
<th>Hole</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>130TB</td>
<td>131TB</td>
</tr>
<tr>
<td>B</td>
<td>1GB</td>
<td>1TB</td>
</tr>
<tr>
<td>C</td>
<td>4KB</td>
<td>4GB</td>
</tr>
</tbody>
</table>

Size ??

Looking for this

Holes

EAP
### Size Distributions

<table>
<thead>
<tr>
<th>Hole</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>130TB</td>
<td>131TB</td>
</tr>
<tr>
<td>B</td>
<td>1GB</td>
<td>1TB</td>
</tr>
<tr>
<td>C</td>
<td>4KB</td>
<td>4GB</td>
</tr>
</tbody>
</table>

- **Safe Stack**
- **Heap**
- **Data**
- **Code**
- **mmap**
- **Unsafe Stack**
- **SIZE X**
- **deallocated**

**Looking for this**

**Holes**

**EAP**
Persistent Allocation Primitive (PAP)

- For each request:
  
  ```c
  ptr = malloc(size);
  ...
  reply(result);
  ```

- Pure persistent primitives rare
- But we can often turn *ephemeral* into *persistent*
  - Keep the connection open
  - Do not complete the req-reply
### Size Distributions

<table>
<thead>
<tr>
<th>Hole</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>130TB</td>
<td>131TB</td>
</tr>
<tr>
<td>B</td>
<td>1GB</td>
<td>1TB</td>
</tr>
<tr>
<td>C</td>
<td>4KB</td>
<td>4GB</td>
</tr>
</tbody>
</table>
Size Distributions

<table>
<thead>
<tr>
<th>Hole</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>130TB</td>
<td>131TB</td>
</tr>
<tr>
<td>B</td>
<td>1GB</td>
<td>1TB</td>
</tr>
<tr>
<td>C</td>
<td>4KB</td>
<td>4GB</td>
</tr>
</tbody>
</table>
## Size Distributions

<table>
<thead>
<tr>
<th>Hole</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>130TB</td>
<td>131TB</td>
</tr>
<tr>
<td>B</td>
<td>1GB</td>
<td>1TB</td>
</tr>
<tr>
<td>C</td>
<td>4KB</td>
<td>4GB</td>
</tr>
</tbody>
</table>

- **Safe Stack**
- **Heap**
- **Data**
- **Code**
- **mmap**
- **Unsafe Stack**

**Holes**

- **A**
- **B**
- **C**

**High addr.**

**Low addr.**

**EAP**

**SS at Heap + Y**

**PAP**
So we need

- A way to effect large allocations repeatedly
- A way to detect whether they failed
Here is what we do

• A way to effect large allocations repeatedly
• A way to detect whether they failed

```c
ngx_event_accept(ngx_event_t *ev) {
...
  ngx_connection_t *lc = ev->data;
  ngx_listening_t *ls = cl->listening;
  ...
  c->pool = ngx_create_pool(ls->pool_size, ev->log);
  ...
}
```

• When server is in quiescent state
  • Taint all memory
  • See which bytes end up in allocation size
Here is what we do

• A way to effect large allocations repeatedly
• A way to detect whether they failed

Options
• Direct observation (most common)
  • E.g., HTTP 200 vs. 500
• Fault side channels
  • E.g., HTTP 200 vs. crash
• Timing side channels
  • E.g., VMA cache hit vs. miss
Examples

• Nginx
  • Failed allocation: Connection close.

• Lighttpd
  • We crash both when
    • allocation fails (too large) and
    • succeeds (but allocation > than physical memory)
  • But in former case: crash immediately
  • In latter case, many page faults, takes a long time
Assumption

**Memory overcommit:**

• OS should allow (virtual) allocations beyond available physical memory
  • Common in server settings
  • Required by some applications:
    • Redis, Hadoop, virtualization, etc.

• However, even when disabled:
  • Allocation oracles still possible
  • But attacker has to bypass overcommit restrictions
Conclusion

• Implementing safe stacks without pointers to it might not be trivial

• ASLR can be weakened by using Thread Spraying and Allocation Oracles

• Proper isolation can mitigate these attacks