Secure Linking in the CheriBSD Operating System

Alexander Richardson, Robert N. M. Watson
University of Cambridge

PriSC 2019
13 January 2019
Outline

• A little about the CHERI architecture
• What do we mean by secure linking in the CHERI context?
• CHERI pure-capability protection before secure linking
• Improvements made by secure linking
• What more could be done?
Pointers today

• Implemented as integer virtual addresses (VAs)
• (Usually) point into allocations, mappings
  • Derived from other pointers via integer arithmetic
  • Dereferenced via jump, load, store
• No integrity protection – can be injected/corrupted
• Arithmetic errors – out-of-bounds leaks/overwrites
• Inappropriate use – executable data, format strings

➢ Attacks on data and code pointers are highly effective, often achieving arbitrary code execution
Protection model: 256-bit capabilities

CHERI capabilities extend pointers with:

- **Tags** protect capabilities in registers and memory:
  - Dereferencing an untagged capability throws an exception
  - In-memory overwrite automatically clears capability tag
- **Bounds** limit range of address space accessible via pointer
- **Permissions** limit operations – e.g., load, store, fetch
- **Sealing** for encapsulation: immutable, non-dereferenceable
**Architecture: 128-bit compressed capabilities**

- **Compress bounds** relative to 64-bit virtual address
  - Floating-point bounds mechanism constrains bounds alignment
  - Security properties maintained (e.g., provenance, monotonicity)
  - Formats for sealed, non-sealed capabilities invest bits differently
  - Strong C-language support (e.g., for out-of-bound pointers)
CHERI enforces protection semantics for pointers

- **Integrity** and **provenance validity** ensure that valid pointers are derived from other valid pointers via valid transformations; **invalid pointers cannot be used**
- **Bounds** prevent pointers from being manipulated to access the wrong object
- **Permissions** limit unintended use of pointers; e.g., $W^X$ for pointers
- **Monotonicity** prevents pointer privilege escalation – e.g., broadening bounds

- However, bounds and permissions must be **initialized correctly** by software – e.g., stack allocator, heap allocator, **dynamic linker**
```
int x;
int y;

int main(int argc, char** argv) {
    int *ptr = &x;
    *ptr = 1;  // this is fine
}
```
Example: protection for global variables

```c
int x;
int y;

int main(int argc, char** argv) {
    int *ptr = &x;
    *ptr = 1;  // this is fine
}
```
Example: protection for global variables

```c
int x;
int y;

int main(int argc, char** argv) {
    int *ptr = &x;
    *ptr = 1;  // this is fine

    // address is the same as &y
    int *ptr2 = &x + 1;
    *ptr2 = 2;  // what happens here?
}
```
Example: protection for global variables

```c
int x;
int y;

int main(int argc, char** argv) {
    int *ptr = &x;
    *ptr = 1;  // this is fine

    // address is the same as &y
    int *ptr2 = &x + 1;
    *ptr2 = 2; // what happens here?
}
```

Most architectures permit storing to y using a pointer derived from x.
Example: protection for global variables

```c
int x;
int y;

int main(int argc, char** argv) {
    int *ptr = &x;
    *ptr = 1;  // this is fine
    // address is the same as &y
    int *ptr2 = &x + 1;
    *ptr2 = 2; // what happens here?
}
```

Using CHERI we can ensure that a write to y via a pointer to x always fails. If the initial bounds were set correctly

Most architectures permit storing to y using a pointer derived from x
Overall goal: reducing available privilege

• By privilege we mean the **memory accessible at a given time** in the program’s execution
  • For now we ignore file system and network access rights. This kind of sandboxing can be managed differently (e.g. by using Capsicum)
• In a conventional architecture privilege is all memory **mapped as accessible by the MMU**
  • **Every integer is also a valid pointer** and can therefore be used to access memory.
  • ASLR makes arbitrary accesses more difficult but does not prevent them.
• With CHERI privilege is the set of **all capabilities transitively reachable** from the current register contents.
  • The **MMU can further restrict** accessible memory (but is not essential).
  • The CheriBSD kernel ensures that memory management APIs can’t break capability monotonicity.
CHERI pure-capability linkage design goals

By reducing the amount of privilege available, we can achieve the following:

• **Completely eliminate out-of-bounds memory accesses for global variables**
  • Memory outside of the current DSO should be inaccessible (except for exported symbols)

• **Even stronger protection against control-flow hijacking**
  • CHERI hardware already prevents arbitrary jumps
  • Linker support can reduce the number of accessible code capabilities

• **Reduce the size of the TCB**
  • Compiler code-generation bugs can’t break the overall security model since we don’t rely on compiler-inserted checks
  • However, compiler and static linker are **partially trusted** to create an ELF file with a valid symbol table and relocations to be processed by kernel ELF loader and dynamic linker
  • Only the runtime linker and the kernel should are fully trusted but not libc.so, etc.
CHERI pure-capability code without secure linkage

- Capabilities to global variables are derived by using the virtual addresses from the GOT as an offset into **program counter capability ($pcc)** or **default data capability ($ddc)**.

- MIPS globals pointer ($gp) used to find GOT by indexing into $ddc.
Capabilities to global variables are derived by using the virtual addresses from the GOT as an offset into **program counter capability ($pcc)** or **default data capability ($ddc)$**.

- MIPS globals pointer ($gp$) used to find GOT by indexing into $ddc$.

- $ddc$ spans the whole address space and is writable!

This means an attacker can write anywhere (including code)!

Virtual address of &myint

Accessible!
Capabilties to global variables are derived by using the virtual addresses from the GOT as an offset into program counter capability ($\text{pcc}$) or default data capability ($\text{ddc}$).

- MIPS globals pointer ($\text{gp}$) used to find GOT by indexing into $\text{ddc}$.

Virtual address of &\text{bar}  
Virtual address of &\text{myint}  

Stack ($\text{csp}$):

\begin{align*}
\text{stackframe #1} & & \text{.got ($\text{ddc} + \text{gp}$)} \\
\text{stackframe #2} & & \text{.data ($\text{ddc}$)}
\end{align*}

\text{.text ($\text{pcc}$/\text{cra}$)}:

\begin{align*}
\text{int foo()} & \{ \\
& \text{return bar();} \\
& \}
\end{align*}

\begin{align*}
\text{int bar()} & \{ \\
& \text{return myint;} \\
& \}
\end{align*}

\text{virtual address of &\text{myint}}

\text{secret\_func() in libsecret.so}

$\text{pcc}$ spans the whole address space and is executable!
This means an attacker can jump anywhere (including data)!

$\text{ddc}$ spans the whole address space and is writable!
This means an attacker can write anywhere (including code)!

\text{secret\_key (libsecret.so)}

Accessible!
Bounds on global variables without linker support

- Capabilities to global variables are derived by using the virtual addresses from the GOT as an offset into $ppc or $ddc
- Bounds on global variables are implemented in the compiler by adding CSetBounds instructions for global variables as is done for stack allocations
  - The executing code still has access to ambient capabilities that need to be bounded correctly → compiler code generation bugs can result in excessive privilege
  - Furthermore, this only works if the size of a variable is known
  - Can use various hacks to almost make it work for external symbols
- This model (mostly) works but has various limitations
Accessing global variables with linker support

- Existing architectures can just generate any integer value and use that to access a variable.
  - This is not possible with CHERI due to monotonicity and integrity.
- Alternatively they can add a constant to $pc/$gp/toc/etc. in the PIC case (which must be within bounds for CHERI).
- For CHERI all global variable accesses and function calls must load an authorizing capability from a GOT-like table (the captable) even for position-dependent code.
- The static linker emits relocations to initialize capabilities in the globals table that are processed by the runtime linker on program startup.
  - All capabilities must be initialized anyway because non-RAM storage cannot save tags. This initialization is equivalent to relocating pointer values by the load address in PIE.
  - PIE increasingly the default for ASLR so this adds no new overhead from CHERI compared to commonly on by default vulnerability mitigation techniques.
- Every function needs a capability for the globals table ($cgp) on entry
PC-relative linkage model

- $\textit{cgp}$ is generated by \textbf{adding a static link-time constant to $\textit{pcc}$}.
  - This means $\textit{ddc}$ can now be NULL.

- **Advantages:**
  - $\textit{cgp}$ can be generated within function so no need to pass as it as an (implicit) argument.
    - This means function pointers can point directly to the function and do not need a trampoline that generates $\textit{cgp}$
  - Very similar to existing MIPS code generation (same number of instructions). Therefore a good model for fair benchmarks between pure-capability and legacy MIPS code
  - More efficient in contemporary architectures with pc-relative loads/AUIPC

- **Disadvantages:**
  - $\textit{pcc}$ must grant access to both the current function and the table of capabilities (i.e., .text and .captable section) and requires at least LOAD_DATA and LOAD_CAP permissions on $\textit{pcc}$
  - An attacker with arbitrary code execution could jump to any instruction within the current DSO
PC-relative linkage model

- All privilege held in three registers: **stack pointer ($csp)**, **program counter ($pcc)** and **return capability ($cra)**. The **globals pointer ($cgp)$** is generated from $pcc$.

- Since $\text{ddc is now NULL}$ only globals listed in the captable are accessible.

```
int foo() {
    return bar();
}

int bar() {
    return myint;
}
```

```
int foo() {
    return bar();
}
```

```
myint = 2
```

libsecret.so

```
secret_func() in libsecret.so
```

```
secret_key (libsecret.so)
```
PC-relative linkage model

- All privilege held in three registers: stack pointer ($csp), program counter ($pcc) and return capability ($cra). The globals pointer ($cgp) is generated from $pcc.

- Since $ddc$ is now NULL only globals listed in the captable are accessible.

Stack ($csp$):

.Text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):

.text ($pcc$/$cra$):
PC-relative linkage model

- All privilege held in three registers: **stack pointer** ($csp), **program counter** ($pcc) and **return capability** ($cra). The **globals pointer** ($cgp) is generated from $pcc.

- Since **$ddc** is now **NULL** only globals listed in the captable are accessible.

Stack ($csp):

- Stackframe #1
- Stackframe #2

.text ($pcc/$cra)

- int foo() {
  return bar();
}
- int bar() {
  return myint;
}

Can update $pcc to point to bar() if bar() is in the same DSO, otherwise inaccessible.

Inaccessible (different DSO)

secret_func() in libsecret.so

Can only access globals that are available in current .captable

myint = 2

secret_key (libsecret.so)

Inaccessible (different DSO)
PLT linkage model

- **$cgp** must be set **correctly on function entry** and is a caller-save register
  - This value can remain the same for calls within a library

- **Advantages:**
  - Saves three instructions on function entry to generate $cgp
  - $pcc is bounded to the current function
  - An attacker with arbitrary code execution only has access to capabilities in the captable

- **Disadvantages:**
  - $cgp must be set correctly by the caller or a PLT stub (which adds four instructions including two memory loads)
  - Function pointers cannot point to the function but a trampoline that sets up $cgp
    - This is required to call from a context with a different $cgp (e.g., UNIX signal handlers).
    - This makes it harder to ensure they are **globally unique** (required by C standard).
PLT linkage model

- All privilege held in four bounded registers: $csp$, $pcc$, $cgp$ and $cra$
- $pcc$ is bounded to only the current function.

Stack ($csp$):

.text ($pcc$/$cra$):

.instr foo() {
    return bar();
}

.instr bar() {
    return myint;
}

.secret_func() in libsecret.so

.data ($ddc = NULL$):

myint = 2

secret_key (libsecret.so)

Can still return using $cra
PLT linkage model

- All privilege held in four registers: $csp, $pcc, $cgp, $cra
- $pcc bounded to only the current function

Stack ($csp):

.text ($pcc/$cra)

.int foo() {
    return bar();
}

.int bar() {
    return myint;
}

.secret_func() in libsecret.so

.captable ($cgp)

&bar

&local_secret1

&local_secret2

&myint

.data ($ddc = NULL)

local_secret1

myint = 2

local_secret2

secret_key
(libsecret.so)

All globals in the .captable section are accessible!
PLT linkage model

- All privilege held in four registers: $csp, $pcc, $cgp, and $cra
- $pcc bound to only the current function

Stack ($csp):

Called function can still access caller’s stack frame!

All globals in the .captable section are accessible!

STACK

.text ($pcc/$cra)

.int foo() {
    return bar();
}

.int bar() {
    return myint;
}

.secret_func() in libsecret.so

.data ($ddc = NULL)

.myint

&local_secret1

&local_secret2

&bar

.local_secret1

.local_secret2

myint = 2

secret_key (libsecret.so)
Per-function .captable

- Each function uses a different $cgp \rightarrow$ Privilege granted by $cgp$ is now **minimal**.
- Variables used by other functions are **inaccessible**.
Per-function .captable

• How can we find the correct table?
  • Static linker emits all per-function/per-file tables and concatenates them in a single .captable section
  • Also emits a special special ELF section that contains a mapping from function address to required .captable subset
  • Run-time linker can use this section when creating PLT stubs for exported function or external calls
  • Note: the run-time linker must also insert a PLT stub for every local call since every function needs a different $cgp value
  • Per-function tables will result in duplicate capabilities in the .captable. Some deduplication is possible for functions using the same set of globals.
Beyond basic privilege reduction

- Every library transition stub uses a **return stub** instead of returning to the caller directly.
- This allows switching to a separate stack on function transitions (or bounding and clearing it).
- Could also clear non-argument registers or validate control flow.

```c
void foo() {
    // …
    bar();
    // …
    return;
}
```
Beyond basic privilege reduction

- Every library transition stub uses a **return stub** instead of returning to the caller directly.
- This allows switching to a separate stack on function transitions (or bounding and clearing it).
- Could also clear non-argument registers or validate control flow.

```c
void foo() {
    // ...
    bar();
    // ...
    return;
}
```

**PLT stub for bar()**
- Load $cgp + target $pcc
- Save return $cra and return $csp
- Limit or allocate new stack for bar()
- Allocate return stub and set $cra
Beyond basic privilege reduction

• Every library transition stub uses a **return stub** instead of returning to the caller directly.
• This allows switching to a separate stack on function transitions (or bounding and clearing it).
• Could also clear non-argument registers or validate control flow.

```c
void foo() {
    // …
    bar();
    // …
    return;
}
```

**PLT stub for bar()**

- Load $cgp + target $pcc
- Save return $cra and return $csp
- Limit or allocate new stack for bar()
- Allocate return stub and set $cra

```c
void bar() {
    // …
}
```
Beyond basic privilege reduction

- Every library transition stub uses a **return stub** instead of returning to the caller directly.
- This allows switching to a separate stack on function transitions (or bounding and clearing it).
- Could also clear non-argument registers or validate control flow.

```c
void foo() {
    // …
    bar();
    // …
    return;
}
```

**PLT stub for bar()**
- Load $cgp + target $pcc
- Save return $cra and return $csp
- Limit or allocate new stack for bar()
- Allocate return stub and set $cra

```c
void bar() {
    // …
}
```

**Return stub for bar()**
- Clear stack used by bar() or free the new stack allocated for bar()
- Restore $cra from foo()
- Restore $csp from foo()
- Return back to foo()
Beyond basic privilege reduction

- Every library transition stub uses a **return stub** instead of returning to the caller directly.
- This allows switching to a separate stack on function transitions (or bounding and clearing it).
- Could also clear non-argument registers or validate control flow.

```c
void foo() {
    // …
    bar();
    // …
    return;
}
```

**PLT stub for bar()**
- Load $cgp + target $pcc
- Save return $cra and return $csp
- Limit or allocate new stack for bar()
- Allocate return stub and set $cra

```c
void bar() {
    // …
}
```

**Return stub for bar()**
- Clear stack used by bar() or free the new stack allocated for bar()
- Restore $cra from foo()
- Restore $csp from foo()
- Return back to foo()
Configurable Linkage Policy

• PLT and return stubs are **dynamically allocated** by the runtime linker
• This allows flexible **policy decisions at link-time and at run-time**
• Runtime linker **supports mixing DSOs with different policies**
• We can therefore use different models depending on performance and security goals on a **per-library granularity**
• Linker and compiler flags can change available privilege scope:
  • General ABI selection: `-cheri-cap-table-abi={legacy,pcrel,plt}`
  • Further narrowing of captable scope (this only makes sense with the PLT ABI): `-Wl,-captable-scope={all,file,function}`
• RTLD can read a configuration file with per-library/binary policy:

  /usr/lib/libsecure.so: new-stack,clear-regs
  /usr/bin/more-speed-less-bounds: clear-regs
  /bin/cat: trust-all

  • Basic infrastructure for this exists but not yet fully implemented
Performance (PC-relative ABI)

Impact commonly less than 5% (compared to MIPS)
PostgreSQL initdb 6.8%
Summary

• We fully support dynamic linking with minimal privilege including dlopen() and lazy binding.

• Compiler code-generation bugs cannot be exploited to gain access to inaccessible data.

• Further security goals such stack and register clearing to prevent data leakage can be enabled with a per-library configurable policy.

• It is possible to mix the different modes even within a process to choose a suitable trade-off between security and performance.

• All code is available on GitHub:
  • https://github.com/CTSRD-CHERI/llvm
  • https://github.com/CTSRD-CHERI/clang
  • https://github.com/CTSRD-CHERI/lld
  • https://github.com/CTSRD-CHERI/cheribsd

• To learn more about the CHERI architecture and prototypes:
  • https://www.cheri-cpu.org/
Questions?
What about loading via the target $pcc or $cra?

• In the current implementation this is still possible.
• However, this can be fixed by using the sealed capability mechanism.
  • Pairs of sealed capabilities can be invoked using CCall,
  • CCall unseals the paired capabilities (the data argument is unsealed into $cgp) and jumps to the code.
• We also have an experimental implementation of call-only sealed capabilities that could be used for call targets and return addresses.
Why don’t we just use pairs of capabilities?

• We could do: by using function descriptors
• However, POSIX APIs require $\text{sizeof}($void*) $\equiv$ $\text{sizeof}($void(*)(void))
• Therefore we need indirection: function pointers are non-executable pointers to a pair of capabilities
• This is more-or-less the same as jumping to a stub that loads the pair
  • Can inline the pair in the captable, but this puts pressure on the limited immediate range in the load capability instruction
• Requires kernel changes to handle non-executable capabilities in $\text{sigaction}()$, etc.
• **Note:** We have an experimental function descriptor implementation with slightly different performance characteristics but the same security properties as the PLT model
Function pointers must be unique

- Required by C and C++ standard
- Cannot use the PLT stub as the function pointer since the stub is different in every library that uses that function.
- Chosen solution: The function pointer always resolves to a stub in the library that exports the function.
- Two different relocations for direct call and taking a function pointer:
  - R_MIPS_CHERI_CAPABILITY_CALL: does not need to be unique so can point to the per-DSO PLT stubs.
  - R_MIPS_CHERI_CAPABILITY: When used with STT_FUNC symbol guarantees a unique address (otherwise a direct data reference).
- Lazy binding is not possible for function pointers but still fine for direct calls.