From The Shadows

Guarded Control Stacks on AArch64

David Spickett, Arm/Linaro

FYI

- Code presented is entirely fictional, but inspired by real events.
- Code presented is deliberately bad, but inspired by real “quality”.
- Other security options are available, many of which also mitigate this problem.
- This is a high level / “user space” introduction to GCS, the Architecture Manual has more detail.

Guarded Control Stack

- Checks return address integrity

(which I will show with a demo program)

- Provides an easy and fast way to backtrace

(which I will show by debugging the demo program)

Return Address Corruption

Save Files

- A hypothetical game has binary save files:

Name	8 bytes (7 characters plus null character)
Coins	8 byte unsigned integer

Lack of checksum and string handling would be 2 things to investigate but not on the menu today.

Loading Save Files

```
void read_save_file(const char* save_file, SaveFile* dest) {
  FILE* f = fopen(save_file, "rb");
  if (!f) {
    printf("Save file not found!\n");
    exit(1);
  }

  fread(dest, sizeof(SaveFile), 2, f);
  fclose(f);
}
```

The Normal Case

My name and 99 coins:

```
echo -n -e 'David_S\x00\x63\x00\x00\x00\x00\x00\x00\x00' > savefile
```

Result:

```
./demo savefile
Hello David_S!
You have 99 coins
```

It works, right?

Problems

- Reviewers might say:
  - Why does it not check how many bytes were actually read?
  - Why does it load 2 SaveFile not 1?

```
  fread(dest, sizeof(SaveFile), 2, f);
  fclose(f);
}
```

Attackers won't say anything, but they will try to exploit this...

Evil Save File

```
echo -n -e
'David_S\x00\x63\x00\x00\x00\x00\x00\x00\x00Dr.Evil\x00\x04\xaa\xaa\xaa\xaa\xaa\x00\x00'
> evilsavefile
```

```
$ xxd evilsavefile
00000000: 4461 7669 645f 5300 6300 0000 0000 0000  David_S.c.......
00000010: 4472 2e45 7669 6c00 04aa aaaa aaaa 0000  Dr.Evil.........
```

- One legitimate save file.
- One for Dr. Evil, who is doing so well, he has 187649984473604 coins.

Evil Save File

```
(gdb) run
Starting program: ./demo ./evilsavefile
Hello David_S!
You have 99 coins
Super secret function!
[Inferior 1 (process 3357284) exited normally]
```

The amount of coins == the address of secret_function!

Caveats:

- -fno-stack-protector, because that also prevents this.
- Using GDB to disable ASLR so I don't have to guess the address each time.
- Real attackers would be more sophisticated and not need these helpers.

Evil Save File

First part is written to “dest”

```
00000000: 4461 7669 645f 5300 6300 0000 0000 0000  David_S.c.......
```

```
00000010: 4472 2e45 7669 6c00 04aa aaaa aaaa 0000  Dr.Evil.........
```

Second part overflows into the a caller’s stack frame

Overwrites the stored link register

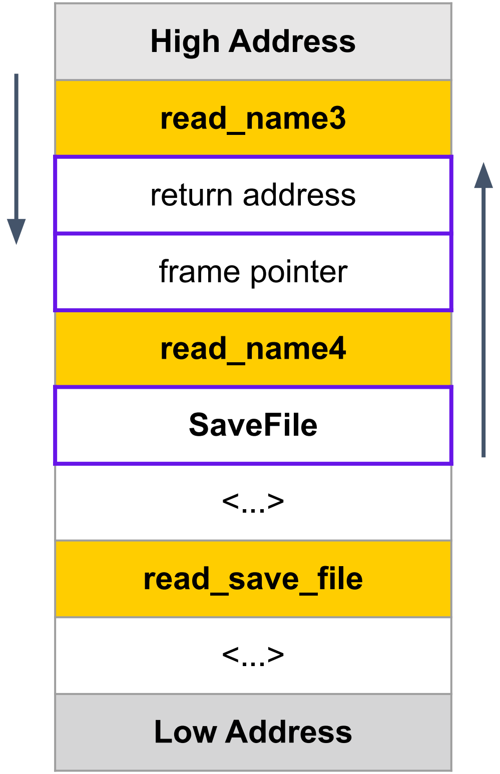
Before And After

```
(lldb) n
-> 70      fread(dest, sizeof(SaveFile), 2, f);
(lldb) bt
  * frame #0: read_save_file
    frame #1: read_name4
    frame #2: read_name3
    frame #3: read_name2
    frame #4: read_name
    frame #5: main
     <...>
(lldb) n
-> 71      fclose(f);
(lldb) bt
  * frame #0: read_save_file
    frame #1: read_name4
    frame #2: read_name3
    frame #3: secret_function
    frame #4: read_name
```

read_name3 -> read_name2 ✅

read_name3 -> secret_function ❌

Stack Overwrite



Could be the start of many types of attacks.

Most relevant is "Return Oriented Programming".

Return Oriented Programming (ROP)

- Attacker takes control of the return address.
- Builds a “chain” of short sequences called “gadgets” in the binary.
- Gadgets do something useful, then end in a control transfer (e.g. return).
- For example, loads from stack into a register.
- Gadgets can be programmatically found in known programs.

Will not cover the mechanics today. The key points are:

- ROP is powerful and can do arbitrary things given enough gadgets.
- ROP relies on the return address being changed.

For more info see: https://llsoftsec.github.io/llsoftsecbook/#return-oriented-programming

Other Options Are Available

Many ways to prevent this specific exploit type:

- nodiscard on fread return value, pay attention to warnings
- Banning “unsafe” C functions (for some definition of “unsafe”)
- Address Sanitizer (not suitable for production)
- Memory Tagging (Arm v8.5-a)
- Pointer Authentication (signing the return address)
- Layout randomisation (make the attacker guess where functions are)
- Stack Protectors and/or Stack Cookies
- Capabilities? (CHERI)
- Static analysis
- Not using C? 🦀
- …

Not all work all the time, not all are focused on return address integrity.

Guarded Control Stack (GCS)

Guarded Control Stack

- FEAT_GCS, optional from Armv9.3-a.
- Hardware implementation of a "shadow stack".

- Return addresses stored in a protected area of memory, in addition to the normal link register and stack.
- Guarded Control Stack Pointer Register (GCSPR_ELx)

- Branch and link pushes to the control stack.
- Procedure return pops from the control stack.

- Popped return address must match the link register's value.

(other GCS management instructions are available)

Example

```
int fn_a() { return fn_b(); }
int fn_b() { return 1; }
```

- fn_a calls fn_b
- fn_b returns to fn_a
- fn_a returns to its caller

Calling fn_b (GCS Push)

```
 1 fn_a():
 2   stp      x29, x30, [sp, #-16]!
 3   mov      x29, sp
 4   bl       fn_b() <-- PC
 5   mov      w0, #1
 6   ldp      x29, x30, [sp], #16
 7   ret
 8
 9 fn_b():
10   mov      w0, #1
11   nop
12   ret
```

Guarded Control Stack

Address	Value	
N	X	← GCSPR
N-8	?	
N-16	?	

PC = 4, Link Register = X

Returning to fn_a (GCS Pop)

linaro Connect

```
1  fn_a():
2    stp     x29, x30, [sp, #-16]!
3    mov     x29, sp
4    bl      fn_b()
5    mov     w0, #1 <-- LR
6    ldp     x29, x30, [sp], #16
7    ret
8
9  fn_b():
10   mov     w0, #1 <-- PC
11   nop
12   ret
```

Guarded Control Stack

Address	Value
N	X
N-8	**5**
N-16	?

← GCSPR

PC = 10, Link Register = **5**

Link register == GCS value
Return is allowed ✅

Returned from fn_b

```
 1 fn_a():
 2   stp      x29, x30, [sp, #-16]!
 3   mov      x29, sp
 4   bl       fn_b()
 5   mov      w0, #1 <-- PC, LR
 6   ldp      x29, x30, [sp], #16
 7   ret
 8
 9 fn_b():
10   mov      w0, #1
11   nop
12   ret
```

Guarded Control Stack

Address	Value	
N	X	← GCSPR
N-8	5	
N-16	?	

PC = 5, Link Register = **5**

linaro Connect

Returning from fn_a (GCS Pop)

```
1 fn_a():
2   stp     x29, x30, [sp, #-16]!
3   mov     x29, sp
4   bl      fn_b()
5   mov     w0, #1
6   ldp     x29, x30, [sp], #16
7   ret   <-- PC
8
9 fn_b():
10  mov     w0, #1
11  nop
12  ret
```

PC = 7, Link Register = **X**

Guarded Control Stack

Address	Value
N	X
N-8	5
N-16	?

← GCSPR (points to N)

Link register reloaded
About to return

Link register == GCS value
Return will be allowed ✅

Corrupted Return Address


```
 1 fn_a():
 2   stp      x29, x30, [sp, #-16]!
 3   mov      x29, sp
 4   bl       fn_b()
 5   mov      w0, #1
 6   ldp      x29, x30, [sp], #16
 7   ret   <-- PC
 8
 9 fn_b():
10   mov      w0, #1
11   nop
12   ret
```

PC = 7, Link Register = **Y**

Guarded Control Stack

Address	Value
N	**X**
N-8	5
N-16	?

← GCSPR

Link register reloaded
About to return

Y != X
Return will not be allowed ❌

Using GCS

Enabling GCS

- When GCS is first enabled, the control stack is empty.
- You cannot return from the point you enable GCS.

Address	Value	
N	**0**	← GCSPR
N-8	?	
N-16	?	

- Unless you manually push addresses…

Enabling GCS

- Demo enables GCS from main, usually C library would have done this for you.
- Syscall using a macro, so we do not have to execute a ret afterwards.
- Push link register to GCS so we can return to main.

```
void enable_gcs() {
  my_prctl(PR_SET_SHADOW_STACK_STATUS, PR_SHADOW_STACK_ENABLE | PR_SHADOW_STACK_PUSH, 0,
0, 0);
  __asm__ __volatile__("sys   #3, C7, C7, #0, x30\n"  /* gcspushm link register */);
}
```

- The syscall:
  - Allocates memory to hold the Control Stack
  - Points the GCSPR_EL0 to the first entry of the Control Stack

GCS to the Rescue

```
$ ./demo_gcs evilsavefile
Hello David_S!
You have 99 coins
Segmentation fault
```

The exploit was stopped!

A backtrace would be nice though...

GCS For Backtracing

Backtracing

An alternative use case for GCS is backtracing.

Walking the GCS is much easier than figuring out stack frame layouts.

Let’s see what we can get after stopping the exploit!

Signal Handler

- Attach handler to SIGSEGV.

```
struct sigaction act;
act.sa_handler = handler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
sigaction(SIGSEGV, &act, NULL);
```

- Handler can look for si_code = SEGV_CPERR (control protection error) (not done in this demo)

Signal Handler

```c
void handler(int signal) {
  uint64_t *gcspr = get_gcspr();
  printf("gcspr is %p\n", gcspr);

  for (; ; gcspr++) {
    uint64_t entry = *gcspr;
    if (entry == 0) {
      break;
    }
    printf("0x%lx ", entry);
  }
  printf("\n");

  exit(0);
}
```

System register read:
mrs %0, s3_3_c2_c5_1

Increment 8 bytes each time

Eventually get to
the top of the GCS

Exit the program

Exploit backtrace

```
Hello David_S!
You have 99 coins
gcspr is 0xffffb03fffc8
0xaaaacd280bb4 0xffffb06627dc 0xffffb03ff000 0xaaaacd280b48
0xaaaacd280b68 0xaaaacd280cbc
```

What are all these addresses?

llvm-symbolizer to the rescue.

llvm-symbolizer

```
$ llvm-symbolizer \
    --obj=./demo_gcs \
    --adjust-vma=0xaaaaaaaa0000 \
    0xaaaaaaaa0bb4 0xfffff7ffa7dc \
    0xfffff7dff000 0xaaaaaaaa0b48 \
    0xaaaaaaaa0b68 0xaaaaaaaa0cbc
```

Demo program binary

Virtual memory offset

Return addresses

https://llvm.org/docs/CommandGuide/llvm-symbolizer.html

Exploit backtrace

`handler`	-	`main.c:96:17`	(would have been used as `exit`'s return address)
`__end__`	-	`??:0:0`	(`0xffff7ffa7dc`, `__kernel_rt_sigreturn`)
`__end__`	-	`??:0:0`	(`0xffff7dff000`, signal handling cap token)
`read_name2`	-	`main.c:83:65`	(where we were going to return to)
`read_name`	-	`main.c:84:64`	
`main`	-	`main.c:137:3`	

GCS shows the path we were **supposed to** take.

https://docs.kernel.org/arch/arm64/gcs.html#signal-handling

GCS Deployment

Linux Deployment

linaro Connect

Covered in more detail by Steve Capper -
"Guarded Control Stack (FEAT_GCS) for Debian", MiniDebConf Cambridge, October 2024

Highlights:

- Binaries are annotated to indicate compatibility with GCS.
- Custom assembler must be reviewed.
- GCS can be detected at runtime using the CHKFEAT instruction.
    - Which executes as a NOP on unsupported hardware.
- End user must opt in via. glibc tuneable.

Details may have changed, please review the presentation in full if you are interested.

Conclusion

GCS in a Nutshell

- Prevent attacks that corrupt the return address.
- Lightweight backtracing for debug and profiling.
- Most code does not need to change.

How to Try GCS

For today's demo:

- Source code
- Arm FVP 11.28.23 set to v9.5-a, run via shrinkwrap
- Linux Kernel 6.15-rc4 (6.13 minimum)
- LLDB 20 (>= 20 required)

Generally:

- Compilers:
  - Clang 19
  - GCC 15.1
- Glibc 2.41
- GDB is in progress
- QEMU is planned
- Hardware at some point in the future.

linaro Connect 2025

Thank You!