SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Systems Programming for ARM
Laboratory Scripts

Dr. Andy Pomfret

Laboratory 8

Playing with Stacks

8.1 Aims 87
8.2 Buffer Overruns and the Stack 87
8.3 Stacks and Security 88
8.4 Summary and Conclusions 91

8.1 Aims

In this laboratory session, you will:

- Experiment with overflowing storage areas allocated on the stack
- Understand the stability and security implications of guarding against buffer overruns

8.2 Buffer Overruns and the Stack

Use Project→Open Project... to open `Stacks\array\array.uvprojx`.

In this simple project, an array of ten unsigned 32-bit integers is declared. Nothing much else happens.

Some of you will already have seen (in this module and elsewhere!) that overrunning buffers allocated on the stack can have some undesirable side-effects. To start this lab, you'll investigate the effect of inadvertent stack corruption, and also see that it's possible to manipulate program flow by corrupting the stack.

Try writing numbers beyond the end of the array. To do this, you may need to 'trick' the compiler – it is smart enough to trap direct accesses to an array that go beyond its bounds in such a simple program. So rather than trying to write directly to (for example) `array[10]`, try declaring a pointer-to-`uint32_t` and setting it equal to `array`, then assigning to the dereferenced pointer using a command such as `*(ptr+10)= 0;`. You may still get a warning. Remember that in the general case, such as when you pass an array to a function and it degrades to a pointer, the compiler can not make checks like these.

What happens, when you run the program in the debugger, after `main()` finishes? You should notice that writing to one particular location, usually `array[10]` or `array[11]`,

causes problems. Why do you think this is?

(i) The array is allocated on the stack by the compiler, as with most local variables. Writing to one of the elements beyond the end of the array is overwriting the stacked value of the link register. When `main()` finishes, the stacked value of the link register is popped into the program counter to return.

Run your code in the debugger, and with execution paused at the start of `main()`, look at the disassembly window. Notice how the first line of `main()` is

```
PUSH {r4,lr}
```

or something similar. (The exact combination of pushed registers depends on the code you've written.) Following that is likely to be a line like

```
SUB sp,sp,0x28
```

which reduces the stack pointer by 40 bytes (hexadecimal 28). Remember it's a descending stack, so this effectively leaves a space 40 bytes long for the array.

At the end of `main()`, the line

```
ADD sp,sp,0x28
```

reverses that change, effectively discarding 40 bytes from the stack, before finally the function returns using

```
POP {r4,pc}
```

(i) The documentation for the Thumb-2 instruction set makes it clear that whenever the `STM` instruction is used, no matter whether it's used in increment or decrement mode, registers will be stored in ascending order of register number in memory. Therefore, in this full descending stack, while in the body of `main()` the stack will look like the diagram in Figure 8.1. Take a moment to make sure you understand this.

Now try to get the `test()` function to run, without calling it, by overwriting the stacked link register with the address of the function. Test your code and make sure it works. What happens at the end of the `test()` function?

8.3 Stacks and Security

You've seen how manipulating parts of the stack by writing beyond the end of allocated storage can be used to affect program flow. This required you to write code that specifically accessed memory it shouldn't, which isn't a security concern, because anyone who can flash new code to a target can already do anything they like anyway.

The real problem comes when existing code overruns the end of an allocated storage area on the stack. This is known as *buffer overrun*, and is always the result of a bug: either a bug where the code scribbles data over the stack, which is a stability concern, or a bug

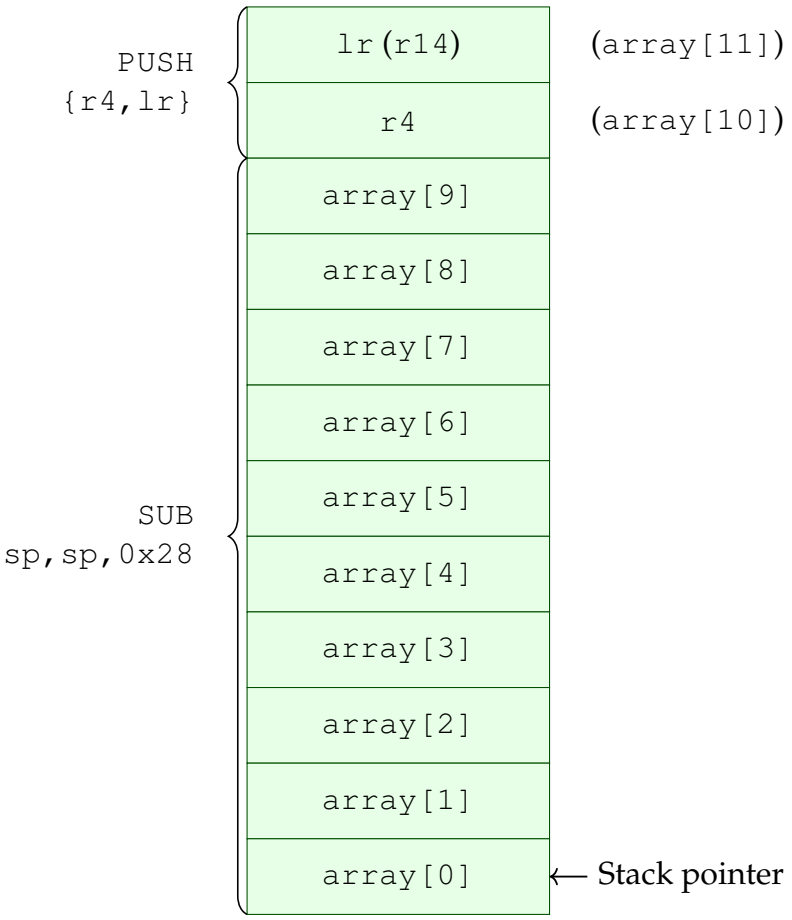

Figure 8.1: The stack state in the body of `main()`

that can be exploited to write specific data into specific locations on the stack, which is a security concern.

Use Project→Open Project... to open `Stacks\password\password.uvprojx`.

This code is emulating the kind of thing you might see in a system that allows a user to enter a password, and then checks it against a stored version to decide whether to allow access to a protected function. In real life the call to `strcpy()` would most likely be a call to `scanf()` or something similar, to accept user input; but although it is possible to configure the boards you're using to accept `scanf()` input from the serial port, most serial consoles doesn't have the ability to send arbitrary ASCII characters, which (as you'll see) would be needed to exploit the buffer overrun bug in this code via the serial port. So you'll just have to pretend that when you change the password attempt string (currently "attempt") you're not changing the code. Hopefully you can appreciate that in the real world, exploiting a buffer overrun bug like this would not require changes to the code.

The `strcpy()` function, in common with most of the standard `C` string functions, copies a

string blindly without any knowledge of the destination buffer size. It's a very common source of buffer overrun bugs, along with `sprintf()`.

Your challenge is to craft a password attempt string that will allow access to the protected function *without* getting the password right. You might find this to be a difficult exercise! Some tips that might be useful:

- You're aiming to overflow the `attempt[]` array to overwrite the stacked link register, as before.
- Begin by using the debugger to look at the disassembly listing, and noting how many registers are pushed to the stack at the start of `main()`.[8]
- Bearing in mind that a **`char`** is one byte long, work out how many bytes you'd need to write to `attempt[]` to overwrite the stored link register. Test your theory by writing something specific to the link register – for example, the string "AAAA" is four ASCII characters with the value 65 (hexadecimal 41) so if you overwrote the link register with this pattern you should expect to see, on stepping through the code with the focus on the disassembly window, that at the end of `main()` the CPU tries to jump to address `0x41414141`.
- Figure out the target address for the jump. You can do this by invoking the debugger, clicking on the `secret()` function in the source window, and looking at the disassembly window. You should see a `PUSH` instruction that starts the function: the address of this instruction is the target address for the jump.
- Set the least-significant bit of the target address to 1.
  Why? Well, in the days of the 32-bit ARM instruction set, instructions were always word-aligned, so the least significant two bits of the link register would always be zero. When the Thumb instruction set was introduced alongside it, with its 16-bit instructions, bit 1 of the link register no longer had to be a zero (16-bit instructions could start on a 2-byte boundary), but bit 0 was still unused – so the ARM designers decided to use this bit to indicate whether the code that was being jumped to should be interpreted as ARM (32-bit) or Thumb (16-bit) code. A 0 meant ARM, and a 1 meant Thumb. This is known as the *Thumb bit*.
  The Cortex-M series use the unified Thumb-2 instruction set, which combines 32-bit and 16-bit instructions into one set. But because technically it's an extension of the Thumb instruction set, not the ARM instruction set, the Thumb bit *must be set* when performing a branch to an address held in a register. If the Thumb bit is not set – in other words, if the least-significant bit of the link register is zero – then the CPU will assume that it is being asked to execute instructions in ARM format, and an immediate hard fault will result.
- When you're constructing your string, you can insert arbitrary bytes using the

[8] When writing your own code in the future, remember that exploiting security bugs like this only requires a disassembly listing – which in turn only needs access to the compiled binary that's running on the device, not the source code. Gaining access to this is usually not very hard.

`\x` escape sequence and a two-digit hexadecimal number. For example, to add a byte with the value 0x3A, use `\x3A` in your string.

- Finally, you need to know that in common with most other modern CPU cores, the Cortex-M4F as configured for the STM32 processor range is *little-endian*. In other words, when storing a value that's more than one byte long, the CPU will store that value with the least-significant byte *first*. So you need to supply the bytes of your target address in reverse order, least-significant byte first.

(i) You will only be able to affect two bytes of the link register. Why? (Think about how `strcpy()` works.) Your attack should still be successful though, because the target address and the original return address are not very different, and share a common top 16 bits.

8.4 Summary and Conclusions

By now, you should:

- Have seen how the stack contains different types of data from different places, mixed together
- Understand the potentially disastrous effects of buffer overrun and stack corruption

If you have any questions about what you have learned, please ask.

Remember to copy your project files to a network drive before you log off.