Chapter 27
Memory Safety Checks

Revision 1.0, October 2025.

Objectives

- Understand the motivation for memory safety checks
- Learn some mechanisms that the compiler or operating uses to protect your program
  Stack canaries, dynamic memory checks, stack frame randomization, and page protection
- Learn about their history
- Understand their limitations

27.1 Motivation

The good news is that memory errors have gotten to exploit. As we saw in Chapter 26, ASLR increased the difficulty of injecting code into a program and of finding the address of useful functions to call in such an attack. In this chapter, we will present a variety of other techniques used by the compiler and operating system to increase the difficulty of exploiting memory errors.

Memory errors apply mostly to the C and C++ programming languages, which provide few safeguards against the dangerous practices that allow such errors. While many programmers still like to write in these language, or need to for compatibility reasons, they add an extra source of vulnerabilities to a program. And this class of vulnerabilities continues to be a major cause of vulnerabilities in the real world.

27.2 Stack Canaries

We return again to our favorite stack smashing example from Figure 1 in the previous chapter and from Chapter 3. The main action of the function is to input a character string from standard input using the dangerous `gets` library function, which has no way to specify the maximum length of the input, making it easy to overflow `buffer`. As we have seen before, the attacker feeds a long input string to this function and `buffer` fills up and overflows in adjacent memory, in this case, overwriting the return address, allowing the attack to control the location that that the program branches to when the return is executed.

Consider the possibility that we might be able to detect when critical values on the stack, like the return address, are overwritten before we try to execute the return instruction. That is the idea behind a mechanism called a *stack*

canary. Stack canaries we introduced by Cowan *et al* in 1997 and the paper that introduced them was formally published in 1998[1].

27.2.1 The Mechanics of Stack Canaries

Extra variables are placed on the stack. These variables are called "canaries" based the way of miners brought canaries down into a mine. The idea was that canaries are more sensitive to poisonous gas than humans, so would provide an advanced warning of a danger before it was fatal to the miners.

```
int foo()
{
  char buffer[100];
  int i, j;
  [on entry: store canary value on stack]
  ...
  gets(buffer);
  ...
  [before return: check canary value]
  return(strlen(buffer));
}
```

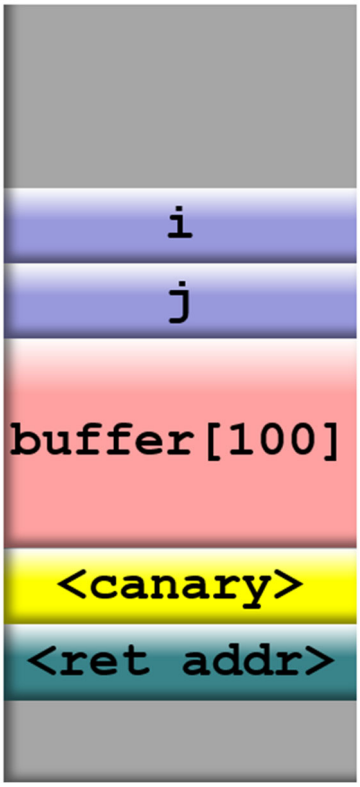

Figure 1: User Program with Canary Code Added, along with Stack Frame with Canary

Stack canaries are placed on the stack in such way that if a buffer on the stack overflows into the return address, the canary variable will also be overwritten. In Figure *1*, we can see the canary value illustrated in yellow on

[1] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks", *7th USENIX Security Symposium,* San Antonio, TX, January 1998.

the stack. The key to a stack canary is to check its value just before the return address is used, that is just before executing the return instruction.

To make this check, extra code is inserted into your program to see if the canary value was modified. We show this extra code (in red) in Figure *1*, however the actual code only appears in the executable (binary) code generated by the compiler and is not visible in the source code. As the programmer, you do not have to do anything special. The only noticeable effect of using stack canaries is a small slowdown in the program's performance. While it is possible to stop the compiler from generating stack canaries and the checking code, this is usually a very bad idea.

If an attack resulted in the canary variable being modified, then the program would throw an exception or exit.

27.2.2 Types of Stack Canaries

There have been several kinds of stack canaries proposed.

Terminator: The simplest is the canary with a zero value. In this case, the canary value is always zero and there is no check at the end of the function. The idea is to prevent any string operation from overflowing into the return address. Since C-style strings are supposed to be null-terminated, the presence of this zero canary would force any string operation to stop before it got to the return address. This is a simple and limited mechanism and not in general use..

Random: The is most commonly used mechanism. is to assign a random canary value to each function each time it is compiled, as was described above. Since each function has its own canary value and since each time the program is compiled, the canaries change, they are difficult to guess or detect.

Random-XOR: A slightly more complicated canary takes the random value and does an exclusive-or with other key information on the stack like the stack frame pointer. This extra step can help to detect tampering with the frame pointer.

27.2.3 Using Stack Canaries

Stack canaries are a real mechanism that is likely available in the compiler that you are already using, such as the open source gcc and clang C or C++ compilers, the Microsoft Visual Studio C or C++ compilers, and many others.

While stack canaries often are enabled by default, they might not be generated for all functions. So, for gcc and clang, it is a good idea to use the `-fstack-protector-all option` when compiling your source file. These

compilers also have the `-fstack-protector` and `-fstack-protector-strong` options, which protect only a subset of the functions in your program. For the Visual Studio compilers, the /GS option turns on stack canaries. You should not have to specify this option as it defaults to on. Other compilers have similar options.

27.2.4 Attacks on Stack Canaries

Attackers will try to combine other techniques with the buffer overflow to find a canary value to accomplish a stack smashing attack. One way in which they try to expose the canary value is to trigger an exception. If, as we described in Chapter 13 on exceptions, your exception handler exposes too much information, like a stack trace, the attacker might be able to see canary value. They can then follow up the exception with an attacker the includes the now-known canary value.

Another way that attackers might try to expose canary values is by trying to read the program's memory. On UNIX systems (which includes Linux and MacOS), `/dev/mem` is a pseudo-file that allows a running program (process) to read and write its own memory using file system read and write system calls. If an attacker can find a Directory Traversal Attack as discussed in Chapter 12, then they could open `/dev/mem` and read the canary values.

While such combined attacks sound esoteric, they are surprisingly frequent in the real world, especially used by highly trained hacker teams. Vulnerability development is often a deliberation and patient process.

27.3 Dynamic Memory Checks

Now we will look at a second technique that the compiler provides to make dynamic (heap) memory references more secure. Errors in accessing heap memory are far too common and often the source of serious program vulnerabilities.

Two of the more serious errors that a programmer can make when dealing with heap memory references are:

Use after free: In this case, the program has deallocated the memory using the C `malloc` function or the C++ `delete` operator but continues to use pointers to the memory that was freed. If the memory that was freed is then reallocated, uses of the old pointer can have unexpected consequences. This errors is common source of real world vulnerabilities.

Buffer overrun or overflow: This error is the heap equivalent to a stack smashing attack. Using a pointer or array index incorrectly can allow the program to read or write beyond the bounds of the allocated memory. Reading beyond the bounds can allow a program to reveal private

information (such as was done in the Heartbleed vulnerability[2]). Writing beyond the bounds can change the program's behavior by writing over other allocated data or disrupt it by overwriting the data structures that keep track of the dynamically allocated memory.

Bad pointers: It is common error to have a pointer with simply the wrong value, either because of a calculation error or improper initialization. Using such a pointer can result in the intended operation happening on almost any location in the program's memory.

Protecting against such errors is based, again, on the compiler automatically inserting extra instructions into your code. This time, these extra instructions keep more careful track of what heap memory is allocated and then check each memory reference to see if it accesses valid memory. Of course, the extra tracking and checking instructions can cause a noticeable slowdown in your code, so this technique most commonly is used during debugging and not included in the released version of the software.

A popular tool for making such checks is the Address Sanitizer[3] that is available to use with either the gcc or clang compilers. To understand this tool and how it is used, we will start with a simple program that has a heap overflow, show how to compiler it to activate the Address Sanitizer functionality, and then look at the report that the Address Sanitizer generates when it encounters a memory error.

We start by looking at the fragment of C code shown in Figure 2. From a quick examination of the code, you can see that amount of memory allocated for the array pointed to by `p` on line 6 is much less than being written in the `for` loop on line 9.

```
06  long *p;
07  p = (long *)malloc(sizeof(long)*10);
08  for (i=0; i<1000; i++) {
09    p[i] = 7;
10  }
```

Figure 2: Fragment of Sample `testmem.c` Program with Heap Memory Overflow

Our first step is to compiler the program with the Address Sanitizer option enabled and then run the program:

```
% gcc -g -Og -o testmem -fsanitize=address testmem.c
% testmem
```

[2] `https://en.wikipedia.org/wiki/Heartbleed`
[3] `https://clang.llvm.org/docs/AddressSanitizer.html`

To get more informative results, we also enable debug information using `-g` and turn of optimization using `-Og` (not `-O0`, which you should never use). When we run the program, Address Sanitizer produces the output we see in Figure 3. While this output looks a bit messy it is a bit clearer to read on a wide screen. And, after using it for a while, the contents become easier to understand. It is worth taking the time to look at this output carefully to see all the details presented.

From this output, we learn two important things. First, we learn what kind of memory access error happened and where it happened. Second, we learn where the memory was allocated.

The first thing we learn is at the top of Figure 3, where we see an "ERROR" notice along with the fact that it was a `heap-buffer-overflow`. Along with this notice, we see address of the instruction that caused the error (the "pc"), the fact that is was a write operation, and that the write size was 8 bytes (which gives you some idea of the type of the variable being written). Most important though, we see the location in the source code (because you compiled the program with debug information, `-g`), line 9 of `testmem.c`. Looking at the code in Figure 2, we see that is the line we would expect to have caused the error.

```
==1080700==ERROR: AddressSanitizer: heap-buffer-overflow on
address 0x607000000150
     at pc 0x559d346ac483 bp 0x7ffdb2c55d20 sp 0x7ffdb2c55d10
WRITE of size 8 at 0x607000000150 thread T0
    #0 0x559d346ac482 in main testmem.c:9
    #1 0x7fdfd28500b2 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x270b2)
    #2 0x559d346ac20d in _start (testmem+0x120d)
0x607000000150 is located 0 bytes to the right of 80-byte region
[0x607000000100,0x607000000150)
allocated by thread T0 here:
    #0 0x7fdfd2b28bc8 in malloc (/lib/x86_64-linux-
gnu/libasan.so.5+0x10dbc8)
    #1 0x559d346ac386 in main testmem.c:7
    #2 0x7fdfd28500b2 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x270b2)
```

Figure 3: Output of Address Sanitizer

The second thing we learn is where in the program the memory that was being accessed (and overflowed) was allocated. From the output, we see that the memory was (unsurprisingly) allocated by malloc, which was called from line 7 of testmem.c. Again, looking at Figure *2*, we see that Address Sanitizer identified the expected line.

In this simple program, you probably could find the error just by inspecting

the code. However, in a real program, with many levels of function calls and hundreds of thousands, if not millions of lines of code, such errors can be impossible to find on your own.

If your program was carefully written and, by this, we mean that you paid compulsive attention to every heap memory allocation and reference, then none of this extra checking would be necessary. However, programs get large and complex. And with a variety of authors, both design and coding errors will creep in. So, if we are going to use languages like C and C++, all of us need this extra help.

We cannot understate how important are tools like Address Sanitizer. They can catch difficult errors long before your software is released, helping to prevent significant vulnerabilities from entering the code. This kind of checking should be use before **every** release of your code.

27.4 Stack Frame Randomization

In Chapter 26 on ASLR, we saw how the operating system can allocate each region of a program at a random address. The compiler can add a level of complexity to allocation by randomizing the order of local variables as they appear on the stack. We intuitively expect the compiler to allocate variables on the stack in the order in which they are declared. Since the compiler generates the code that creates the stack frame and accesses the variable, it can use any order it wants.

Most compilers will, in fact, allocate variables in the order in which they are declared. However, in some cases, that order may be different, perhaps because the compiler wants to group together variables of the same type.

For making the location of variables less predictable, so making it more difficult the understand how to exploit a buffer overflow and craft a successful attack, the compiler can lay out the variables in the stack frame in a random order. Each time the code is compiled, each function's local variables can appear in a different order. In Figure *4*, we see three possible layouts for the stack frame for our stack smashing example code. The first layout is the one that we would expect, followed by two other possible choices that the compiler could make.

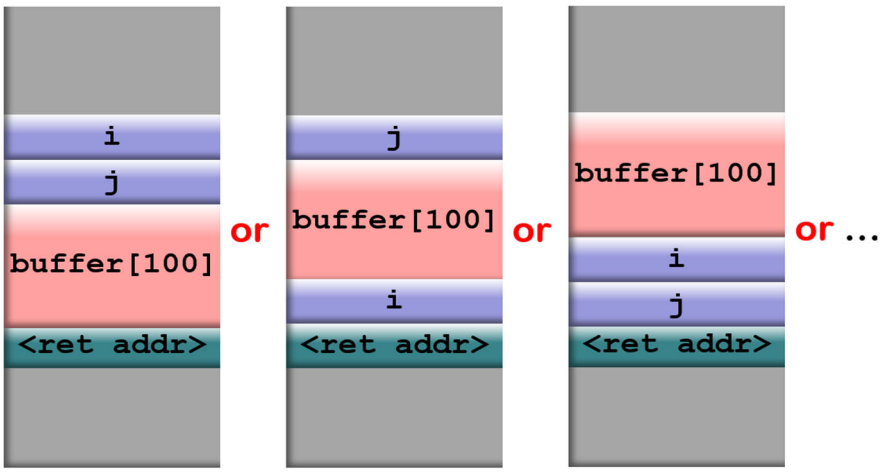

Figure 4: Possible Random Layout of a Stack Frame

One important effect of the two random choices is that they put one or more of the variables between the buffer and the return address. Even without stack canaries (which add another layer of protection), if the buffer overflowed into the return address, it would also overwrite these variables. Such an overwrite could cause enough disturbance in the execution of the program so as to interfere with the attack. In one of our in depth vulnerability assessment activities in the past, we ran into exactly this problem, were we could make a buffer overflow, but could not prevent the program from crashing before the exploit could occur.

Stack frame randomization is available in many modern compilers. For example, it can be activated in gcc and clang by using the `-fstack-shuffle` option

27.5 Page Level Memory Protection

Different parts of the *address space*[4] of your running program (process) have different uses, such as for code or for read-only data or for readable and writable data. And the memory in which your program resides is divided into fixed size chunks called *pages*. Each page is typically allocated to a single purpose, so a page is either all code, or all read-only data, or all readable and writable data.

The access needs of each of these types of pages might be different. For example, the code pages are read and executed by the CPU, read-only data

[4] "Address space" refers the all the memory that is allocated to and addressable by your program.

does not need to be written or executed, and the heap and stack memory is going to be read and written but never executed.

The good news is that the hardware of the computer and operating system work together provide access permissions for each page that can be tailored to their usage needs. The basic access permission types are read, write, and execute. For example, code pages are read and executed by the CPU, so need, at most, read and execute privileges. There is no reason for that part of memory to be writable. Read-only data does not need to be written or executed. And the heap and stack memory is going to be read and written but never executed.

Protecting code pages from being written can prevent attacks that attempt to overwriting existing code with malicious code. Protecting stack and heap pages from executing code can prevent attacks like the original Morris worm, where a buffer on the stack was filled with machine instructions and then the return address overwritten to jump to the instructions in the buffer.

Note that the idea of page level memory protection is not new idea but was one that modern hardware manufacturers were slow to adopt. Back in 1967, the pioneering Multics operating system project at MIT[5] specified that each page of a program's virtual memory should have protection bits to control access to that page. There were three protection bits for each page, read, write and execute. Of course, the operating system cannot dictate such features unless the hardware supports them. So, the Multics group got General Electric (one of the early computer manufacturers) to extend their GE 635 computer to include page level memory protection. With read, write and execute bits, Multics was able to allocate different parts of a program's address space with appropriate protection.

Even though this idea was well understood in the 1960's, current processor manufacturers were slow to adopt it. Security has often been either a low priority or misunderstood area by these companies. Notably, they all lacked an execute permission bit, which means that code could be executed out of the stack or heap, an important technique used in attacks.

Such execute permission bits were added initially in 2001 and finally reached the Pentium architecture in 2003 and 2004. These bits had a variety of interesting names, such as No eXecute, eXecute Never, and eXecute Disable. Once the processors added this feature, operating systems like Linux and FreeBSD added support. Though, for some reason, Microsoft did not add support to Windows until five years later, in 2009.

[5] `https://en.wikipedia.org/wiki/Multics`

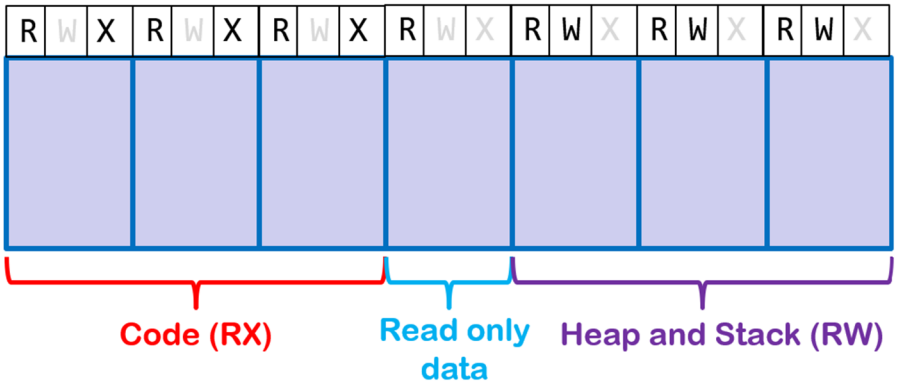

Figure 5: Pages in Memory with Page Protect Bits Set According to Use

Year	Processor	Feature	Year	OS
2001	Intel Itanium	NX (No eXecute)	2004	Linux 2.6.8
2001	ARM v6	XN (eXecute Never)	2004	FreeBSD 5.3
2003	AMD Athlon 64	NX (No eXecute)	2009	Windows XP SP 2
2004	Intel Pentium 4	XD (eXecute Disable)		

Figure 6: Introduction of Page Level Protection for Popular CPUs and Operating Systems

One could say that it was irresponsible of the processor manufacturers to wait more than thirty years to provide such protections. Many preventable attacks occurred in that time interval.

In modern systems, this execute permission is often called W ⊕ X, or write exclusive-or execute. This means that no memory page should ever be both writable and executable. However, as with most things, there are some specials. For example, dynamic compilation or dynamic code generation, often called JIT, just in time compiling, is often used in language interpreters such as for Java. This means that machine instructions are generated at run-time, written into data memory, and then executed from that memory. For these JIT buffers, the pages must be both writable and executable. Special care must be taken when implementing such a feature.

27.6 Summary

- Understood the motivation for memory safety checks
- Learned some mechanisms that the compiler or operating uses to protect your program
  - Stack canaries, dynamic memory checks, stack frame randomization, and page protection

- Learned about their history
- Understood their limitations

27.7 Exercises

1. Research existing vulnerabilities in the National Vulnerability Database and find one that is based on a use-after-free error. Try to find enough details of the vulnerability so that you can describe the coding error that caused the vulnerability.
2. Write a simple C or C++ program that overflows a buffer allocated on the stack. When you execute this program, do you get an error message? Look at the error and see if it sufficiently describes the problem that caused the error.
3. In your favorite operating system, find the system call that allows you to change the memory protect on a page in your program's address space.
4. Why are memory errors most common in C and C++ and not in languages like Java, Python, or Ruby?
5. Rust is a system level programming language like C, with strong protections against memory errors. Investigate what features of Rust make memory errors unlikely.