

Advanced Stack-Smashing and Countermeasures

Understanding how code-injection-attacks exploit the memory of C programs is an important step in learning to write hardened code. This essay will demonstrate a deeper understanding of buffer-overflow attacks, specifically “stack smashing” and will explore the history and industry-use of pertinent countermeasures including ExecShield and StackGuard, while mentioning other relevant defensive concepts along the way.

Many modern code-injection exploits involve the use of buffer overflows. Attackers have developed a myriad of buffer-overflow-style attacks and have used them as an avenue to gain privileged access to victim machines or to cause programs to act in unexpected ways. While computer scientists have been aware of the concept of buffer overflow exploits since at least 1972 (Anderson, 1972), but the style of attack was not widely used until the late eighties. Buffer overflows have been at the core of some notable attacks over time and continue to be a relevant issue today. The notorious 1988 “Morris Worm,” in part, utilized a buffer overflow that was extremely effective against a vulnerable version of the protocol *fingerd* on VAX systems (Roesner, 2017). More recently, hackers have found a buffer overflow vulnerability in PlayStation 2 programs that can grant them the ability to play homemade (“homebrewed”) games on the hacked console. This vulnerability is known as the “PS2 Independence Exploit”. The Morris Worm and PS2 Independence Exploit are just two cases on a long list of destructive buffer overflow attacks. Considering that this type of attack continues to be a pertinent threat, it is important to understand how they work.

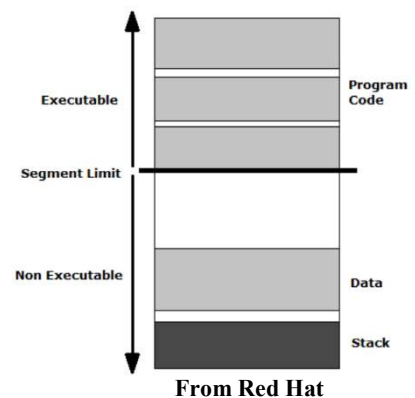
When programs are run, an assortment of variables, function calls, arguments, etc. may be stored in memory, namely on the stack and the heap. If a buffer of a set size is initialized into memory, attackers can place data into said buffer that is larger than its allocated space in memory, thus overflowing it. The memory that sits beyond the end of the buffer can be overwritten by whatever data overflowed the boundaries of the buffer. Functions that involve writing data into an existing buffer in memory but do not also check the bounds of arguments are vulnerable. As such, we often see these attacks targeting C/C++ library functions like `strcpy()`, `scanf()`, and `gets()`. I will focus on stack-based overflow attacks, sometimes called stack-smashing, however it is important to note that there are multiple known overflow-style attacks including stack-smashing, heap-smashing, integer overflows, and format string attacks.

Stack-smashing, which I implemented in Lab 4, is a common form of buffer overflow attack. By overwriting a buffer on the stack, an attacker can overwrite the stack frame’s return address. With the return address compromised, the attacker can redirect the program to a spot in memory where malicious code has been inserted. Commonly, the malicious code that gets executed is shellcode that launches a new shell and command line. Including certain shellcode snippets can grant the attacker access to a shell on the target-machine with root privilege. From here, an attacker can delete/edit files, change machine settings, etc. Adversaries constantly develop methods that make their attacks easier to carry out. One such method is the NOP-sled. Even if the exact location in memory of the malicious code is not known, overwriting the whole stack with 0x90 (No-Operation instructions) will create a “sled” of sorts that will lead to the malicious code. If an attacker can overwrite the vulnerable stack’s return address to force a jump to anywhere within the NOP-sled, the malicious code will run.

As attackers are persistently evolving and finding new ways to overflow buffers, security researchers must similarly work hard to develop up-to-date countermeasures. There are a variety of countermeasures that developers have integrated into their systems and/or compilers that aim to prevent stack smashing and other buffer overflows. Probabilistic countermeasures aim to use randomness to inhibit buffer overflow attackers. Instruction Set Randomization (ISR) is one such probabilistic countermeasure. Normally, the CPU has no way to distinguish expected code from foreign code. ISR encrypts the expected machine code using a randomly generated encryption key. Encryption occurs when instructions are loaded into memory and are only decrypted when they are executed (Younan et al., 2012). Because attackers do not know the randomly generated key, their malicious code will be transformed via decryption into incorrect or unreadable instructions. These decrypted instructions often crash the program, preventing any malicious code from being run. It is important that the encryption key is totally random so that it cannot be guessed in any reasonable amount of time. ISR is not a perfect countermeasure, though. Attackers have found ways around this countermeasure and research notes that ISR can severely hinder performance.

In Lab 4, we saw another example of a probabilistic countermeasure: address space layout randomization (ASLR). ASLR randomizes the location of several key structures and libraries. When the starting address of the stack is randomized, it makes guessing the stack frame's return address nearly impossible. This certainly makes an overflow exploit more difficult to implement, but we also saw in Lab 4 that a brute-force attack, where a script runs the vulnerable program trying every eligible address, can defeat ASLR.

In order to counter brute-force attacks, Red Hat ships a static (non-probabilistic) countermeasure known as "ExecShield." Intel offers a similar countermeasure known as "NX bit." The key innovation that ExecShield brings to the table is making a portion of the stack non-executable. ExecShield divides memory into executable and non-executable segments, putting local machine code into the executable segment but ensuring that data, such as the stack, is located in the non-executable portion (Sidhpurwala, 2018a). With this countermeasure in place, even if an attacker has calculated all relevant addresses and has injected malicious shellcode, the shellcode will not run. Despite labeling this countermeasure as static, ExecShield, in fact, also implements ASLR to randomize the location of key structures and libraries, but this is not its key innovation.



In lab 5, we observed that there are attacks that can easily defeat ExecShield's non-executable stack countermeasure. These "return-oriented" attacks do not actually rely on executing code from the stack. Exploits like the `return-to-libc` attack overwrite a stack's return address to point towards vulnerable libraries which themselves contain instructions that can give the attacker a command line on the victim machine.

To defeat these clever attacks, countermeasures aiming to address these vulnerabilities have been developed. A notably popular and strong countermeasure is the "canary," a countermeasure that has both static and probabilistic versions. In olden times, canaries were brought into mines to act as early warning systems for miners. Because canaries are especially sensitive to poisonous gasses like carbon-monoxide, they would show symptoms of distress before humans, alerting miners to the presence of poisonous gas. In buffer overflow protection, a canary is a similar type of warning system.

Static canaries, also known as terminator canaries, usually contain the four characters `NULL(0x00)`, `CR(0x0d)`, `LF(0x0a)`, and `EOF(0xff)`. These four values terminate most string functions, like `strcpy()`. The canary is placed into the stack before the return address. When overflow occurs and the program reaches the canary in memory, the CPU recognizes these terminator values and cancels the program before it can overwrite the return address. Terminator canaries can be powerful countermeasures but are known to be vulnerable when non-string functions are used to copy buffers (Sidhpurwala, 2018b).

StackGuard, a powerful countermeasure developed by GCC in 1998, operates via probabilistic canary. When implementing a probabilistic canary, a value is inserted onto the stack, just before the return address. The canary value must be probabilistically random in that it cannot be guessed by an attacker in any reasonable amount of time. During program execution, canary protection works by checking to see if the canary value in memory has been changed from its original random value. If there is a change in the canary (the value was overwritten), we know that buffer overflow has likely occurred and the program is safely terminated before return is called. I implemented my own probabilistic canary countermeasure as seen in Figures 1 through 5 below:

In **Figure 1** below, I have modified my `stack.c` file from Lab 4. I initialize a variable called `canary` to `0xdeadbeef`. `0xdeadbeef` is the hex value used in early canary demonstrations and still has meaning in computer science today. After the `strcpy()` function call, but before `return` is executed, I implement a check. The `if`-statement checks to see if `canary` is still equal its initial value. If `canary` is still equal to `0xdeadbeef`, we can say with more confidence than before (there is always a threat) that no overflow occurred. This is the case shown in **Figure 2**. When `badfile` contains just a short packet of data, there is no overwriting on the stack, and the canary check is not triggered.

```
int bof(char *str)
{
    long canary = 0xdeadbeef;
    char buffer[BUF_SIZE];
    /*The following statement has a buffer overflow problem*/
    strcpy(buffer, str);

    if (canary != 0xdeadbeef) {
        printf("Danger detected via modified canary.\n");
        //exit(0);
    }
    return 1;
}
```

Fig. 1

```
[04/30/20]seed@VM:~/535/lab4$ rm badfile
[04/30/20]seed@VM:~/535/lab4$
[04/30/20]seed@VM:~/535/lab4$ echo "noah" > badfile
[04/30/20]seed@VM:~/535/lab4$ ./stack
Returned Properly
[04/30/20]seed@VM:~/535/lab4$
```

Fig. 2

At first, I do not call the `exit()` function because I need to ensure that my stack-smashing attack is still working before I try to implement a canary. When I initialize the variable `canary`, it is placed onto the stack. As such, the offsets that I had calculated for Lab 4 were no longer correct. After the lovely process that is sifting through `gdb`, I was able to get my attack working again. **Figure 3** proves that I got my attack to work while also having the `canary` variable in memory.

```
[04/30/20]seed@VM:~/535/lab4$ ./stack
Danger detected via modified canary.
$
$ sudo ufw status
Status: inactive
$
```

Fig. 3

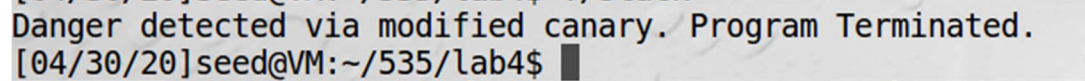
After un-commenting the `exit()` method, the canary works as I designed it. After the vulnerable `strcpy()` runs, my countermeasure recognizes that the value of the canary was changed and, thus, safely terminates

```
if (canary != 0xdeadbeef) {
    printf("Danger detected via modified canary. Program Terminated.\n");
    exit(0);
}
```

Fig. 4

the program. We see this code in **Figure 4**. I also update the `printf` message for more descriptive error handling.

Without this canary, the attack would have been carried through when the return address was reached, like we saw in Figure 2. Instead, the program exits and the user is notified that a modified canary variable indicated a buffer overflow (**Figure 5**).

A terminal window screenshot showing a message: "Danger detected via modified canary. Program Terminated." followed by a prompt "[04/30/20]seed@VM:~/535/lab4\$".

```
Danger detected via modified canary. Program Terminated.  
[04/30/20]seed@VM:~/535/lab4$
```

Fig 5.

Importantly, note that the canary countermeasure that I implemented is not truly probabilistic. I use the value `0xdeadbeef` to make my countermeasure easy to understand and as a nod to the historical significance of that particular hex value. An adversary could easily guess this canary value. In practice, the canary value should be randomly generated at runtime. StackGuard enhances the `crt0` library to choose a set of random canary words at the time the program starts (Cowan et al., 1998). Another major advantage that StackGuard provides over my implementation, which places the canary in the vulnerable file itself, is that the canary is placed during function prologue and checked during the program epilogue. StackGuard automatically injects the code that carries out these tasks during compilation. This means that attackers cannot directly manipulate the logic that handles the canary.

Though I have mentioned several variations of stack-smashing including shellcode injection, the brute-force method, and return-oriented exploits, each is an approach that the computer security community is aware of and working to build defenses against. That being said, it is most important to remember that ASLR, SRI, ExecShield, and StackGuard all have vulnerabilities as well. StackGuard is a leader in its industry space, yet does nothing to combat heap-smashing exploits. Avoid using `strcpy()` without checking the sizes of your arguments. `strncpy()` is a similar function that also takes a bounds-checker argument. I suggest that the best practice is to use a combination of these industry countermeasures and never assume that a program is totally secure.

Works Cited

- Anderson, J. P. (1972). Computer Security Technology Planning Study. *Electronic Systems Division*, 2. Retrieved from <https://web.archive.org/web/20110721060319/http://csrc.nist.gov/publications/history/ande72.pdf>
- Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., ... Zhang, Q. (1998). StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *USENIX*, 7, 4. Retrieved from https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowcan.pdf
- Roesner, F. (2017, August). Software Security: Buffer Overflow Attacks. Retrieved May 1, 2020, from <https://courses.cs.washington.edu/courses/cse484/17au/slides/cse484-lecture3-au17.pdf>
- Sidhpurwala, H. (2018, July 25). Security Technologies: ExecShield. Retrieved from <https://access.redhat.com/blogs/766093/posts/3534821>
- Sidhpurwala, H. (2018, August 20). Security Technologies: Stack Smashing Protection (StackGuard). Retrieved from <https://access.redhat.com/blogs/766093/posts/3548631>
- Younan, Y., Joosen, W., & Piessens, F. (2012). Runtime countermeasures for code injection attacks against C and C programs. *ACM Computing Surveys*, 44(3), 1–28. doi: 10.1145/2187671.2187679