

Introduction

The program stack has many uses in a C program, including parameter passing, local variable storage, temporary register save space, etc. The interactions between these functions, along with careless programming, can sometimes lead to security vulnerabilities in a program. In this laboratory you will create a malicious stack-based exploit for a vulnerable program¹. This looks like a long and scary lab, but it is not that difficult. Most students complete it within 3 hours.

Let's pretend that an embedded system is connected to the Internet and is running a secure web server. You have access to the source code of the web server software and notice the following code fragment:

```
#include <stdio.h>

int gPasswordEntered = 0;

void addToHtml(const char *str)
{
    // Function to display string to user's web browser.
    // Leave this stubbed out.
    ;
}

void printUserWelcome(const char *userName)
{
    char buf[100];

    sprintf(buf, "Welcome to the web site, %s", userName);
    addToHtml(buf);
}
```

The `gPasswordEntered` global variable is set to 1 when the user successfully authenticates himself to the web server using a secret login and password. We are going to force this global variable to be 1, thus gaining access to the secure server without proper authentication.

The `addToHtml()` function displays the string passed to it in the user's web browser. The actual contents of this function are not important hence we represent it as a stub².

The `printUserWelcome()` function accepts a string representing the user's name and then prints a welcome message. Elsewhere in the code, the user is asked for login/password authentication unless `gPasswordEntered` is non-zero, indicating the user has already authenticated himself.

This `printUserWelcome()` function has a serious vulnerability. It uses a 100-character buffer for storing the final string to be displayed, but it does not consider how long the string represented by the `userName` parameter can be. If someone enters a user name that is "too long", it will overflow beyond the end of allocated space for the local variable `buf`. Where is this local variable? On the stack!

The Attack

Your goal in this laboratory is as follows (read this over until it's perfectly clear):

Call the `printUserWelcome()` function with a constant string which will cause the `gPasswordEntered` global variable to have a non-zero value when the function returns.

¹Although the real purpose of the lab is to have you practice your assembly language programming and improve your knowledge of stack operations.

²A *stub* is a function that does nothing for now but is destined to be completed later.

Your attack code for demonstration purposes will be as follows:

```
extern void printUserWelcome(const char *s);
extern int gPasswordEntered;

int main(void)
{
    printUserWelcome("Harold"); // Replace 'Harold' with exploit string

    return gPasswordEntered; // Return value in R0
}
```

The Rules

- You must use exactly the code shown in the two code fragments above. Use two separate C files for the “web server” code and the attack code.
- The only part of the code that you are allowed to change is the string “Harold”, and it must only be replaced with another constant string. You may not add, delete, or otherwise modify the code shown above.
- Your attack must allow the `main()` function to proceed normally after `printUserWelcome()` returns. The return value from `main()` (stored in register R0) will indicate whether or not your attack was successful (0 means you failed).

The Approach

Learn

1. Compile both C files shown above (`lab3a.c` and `lab3b.c`) into a complete program:

```
arm-elf-gcc -o lab3.elf -O3 -march=armv4t -mcpu=arm7tdmi -Wall lab3a.c lab3b.c
```

2. Use the `arm-elf-objdump` tool introduced in the previous laboratory to:
 - study the assembly code of the `printUserWelcome()` function
 - locate the memory address in `main()` which is the first address to execute after `printUserWelcome()` returns
 - locate the memory address of the `gPasswordEntered` global variable (`arm-elf-nm` might be easier)
3. Construct a sketch of the *stack frame* of the `printUserWelcome()` function. That is, what does the stack look like in memory just before `sprintf()` is called? Use numbers to “dimension” your sketch, i.e., how many bytes here, how many bytes there, etc.
4. Use the GDB debugger to single-step through your program one assembly instruction at a time. You should *step over* (instead of *step into*) the call to `sprintf()` as it will be very long and not relevant to today’s laboratory. In terms of GDB commands, use the ‘next’ command to step over a function call, use the ‘step’ command to step into a functional call.
5. In GDB, make a careful note of the value of the stack pointer (R13) immediately upon entry to the `printUserWelcome()` function. If you are to satisfy The Rules (i.e., the `main()` function continues to execute normally when `printUserWelcome()` returns) your exploit will have to ensure that R13 has exactly this value when it returns to `main()`.

Prepare

You should have noticed the following:

- The return address (i.e., the link register) in `printUserWelcome()` has been stored on the stack. If we can overwrite this value on the stack, we can “return” to any address we want rather than returning to `main()`.
- If you pass a user name string to `printUserWelcome()` that is “too long”, you can indeed overwrite the value of the link register stored on the stack.

We now have a mechanism for running arbitrary code on this web server: overwrite the stack so that when `printUserWelcome()` returns, i.e., restores the saved link register from the stack, it actually jumps to the address of code that we supply. This code will set `gPasswordReset` to a non-zero value.

So how can we inject this arbitrary code into the web server? Put it into the user name string passed to `printUserWelcome()`. Code, after all, is nothing more than a sequence of bytes.

The sequence of events will be as follows:

1. `printUserWelcome()` is called with an overly-long “attack string” that contains malicious code as well as the address of this malicious code.
2. The address of the malicious code is carefully positioned within the string so that it overwrites the saved value of the link register when `sprintf()` executes.
3. The malicious code sets `gPasswordReset` to a non-zero value.
4. The malicious code sets the value of the stack pointer `R13` to the correct value for returning to `main()`.
5. The malicious code returns to the correct address within the `main()` function to simulate a successful return from `printUserWelcome()`. It’s OK if `addToHtml()` is not called as part of the exploit.

Design

It is time to write the malicious code. First, write a few lines of independent assembly code (independent from the two C functions we’re currently working with) that:

1. Sets `gPasswordReset` to a non-zero value. You know the address of this global variable from your studies above.
2. Sets `R13` to a constant that is the correct value for the contents of the stack pointer `R13` upon returning to `main()`. You know this value from your GDB single-step session.
3. Sets `R15` (the PC) to the correct return address in `main()`.

Assemble your code using the `arm-elf-as` assembler. Inspect the 32-bit numbers that represent your malicious code using the `arm-elf-objdump` program.

Now we stumble upon a complication. We will take the 32-bit numbers that represent our program and store them in a C string for passing to `printUserWelcome()`. But C strings are terminated by a null (i.e., 0) byte. Any 0 bytes in any one of your instructions (each of which is 4 bytes) will prematurely terminate the C string – `sprintf()` will stop copying bytes from your string to the local variable `buf[]` at the first 0 byte. You must therefore ensure that there are **NO ZERO BYTES IN ANY ONE OF YOUR INSTRUCTIONS!** How? Modify your code a little bit.

When you are satisfied that your code will work, use the `arm-elf-objdump` program to obtain a final listing of all 32-bit numbers that represent your program. You can encode these numbers in a C string using the `\x` escape sequence for placing hexadecimal constants in C strings. For example, the 32-bit instruction `0xE8123456` can be placed into a C string as “`\x56\x34\x12\xE8`” (remember? little-endian?)

Finally, to construct your final attack string you will need to remember that:

- Instructions must be aligned to 4-byte boundaries. The starting location of your attack string in the `buf` local variable might not be properly aligned and you may need to put some dummy bytes before your code.
- You must very carefully also encode the 32-bit address of your malicious code within the string so that it lands directly on top of the saved link register. You cannot be even 1 byte off!

- The length of the string may very well affect the address of `gPasswordReset` and the address of `main()`. You may want to first determine exactly how long the attack string must be, then compile the code with an arbitrary string of this length and compute the necessary constants before writing your final version of the attack code.

Note that C allows you to concatenate constant strings together just by writing them side by side. The C code:

```
printf("abc" "def")
```

is completely equivalent to:

```
printf("abcdef")
```

This feature allows you to construct your attack string in a more readable way:

```
...
"\x04\x30\xA0\xE1" // mov R3,R4      (0xE1A03004)
"\x04\x50\x86\xE2" // add R5,R6,#4  (0xE2865004)
...
```

Integrate and Test

Put it all together, compile your code, then use GDB to step through it. Modify your attack string until it works.

Deliverables

Show me your successful attack before the end of the laboratory period. Graduate students can e-mail their solution to me by Thursday's class.

Additional Notes

- How would you rewrite `printUserWelcome()` to eliminate this security vulnerability?
- How would a "Harvard Architecture" (separate code and data spaces, like the Atmel AVR's) frustrate this type of attack?
- How would your attack be frustrated by a stack that was located much further down in lower memory?
- The type of attack demonstrated in this laboratory is known as a "stack smashing" attack using a "buffer overflow" and is well described at:

http://en.wikipedia.org/wiki/Buffer_overflow

and at:

<http://doc.bughunter.net/buffer-overflow/smash-stack.html>

- Attempting to exploit a vulnerability in an actual, functioning computer system (even if unsuccessful) can land you in lots of hot water. This is true even if you are just trying to be helpful in demonstrating to the system administrator that a vulnerability exists. People go to jail for this kind of thing. So don't do it. See the following article³:

<http://www.cerias.purdue.edu/weblogs/pmeunier/policies-law/post-38>

³Currently unavailable, but Google has cached it. Search for 'pmeunier'.

- Some people argue that open-source software is especially vulnerable to security attacks since malicious people can study the code and look for vulnerabilities. I disagree. The fact that people are free to study the code means that vulnerabilities can be identified quickly and fixed quickly. In addition, automated vulnerability-finding programs can be applied to software for which source code is available. The Linux security mailing lists, for example, often contain disclosures of security vulnerabilities *for which no known exploits have been found* (a vulnerability does not mean an actual exploit is feasible). Microsoft Windows vulnerabilities, by contrast, are mostly identified by virtue of an exploit.

Closed-source code most likely has no-one looking for vulnerabilities thus when malicious people do find them, they can exploit them and know that it will take a long time before the vendor publishes a patch and distributes that patch to its customers.