Rwitam Bandyopadhyay

(Due September 21 2021, 11:59 PM)

1 vulnerable 1

1.1 Briefly describe the behavior of the program

The program takes a command line argument as a "string" input. It then proceeds to copy the same string onto a 100-byte buffer in memory.

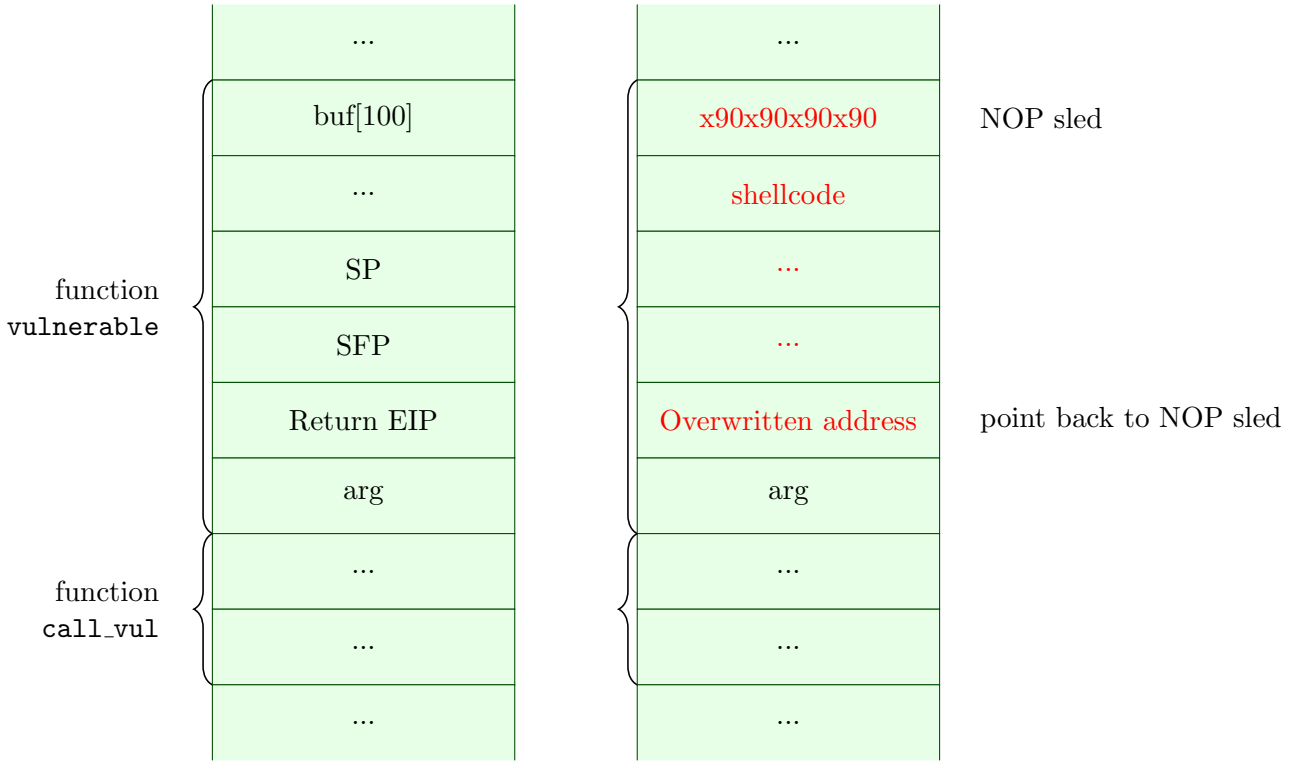
1.2 Identify and describe the vulnerability as well as its implications

The string argument is copied onto the buffer using the following line of C code.

```
strcpy(buf, arg);
```

The C library function `char *strcpy(char *dest, const char *src)` copies the string pointed to, by `src` to `dest`. It does not perform any sort of bounds checking, and this is where extra data can get copied to the destination address and might lead to stack/heap corruption.

1.3 Discuss how your program or script exploits the vulnerability and describe the structure of your attack



So the basic concept for this vulnerability was to overflow the 100 byte wide character buffer, and to extend into the saved EIP for the previous stack frame. The buffer payload is carefully crafted to contain a shellcode, padded on both sides with NOP sleds. The address to place on the "Return EIP" could be anywhere in that NOP sled, or could point to the shellcode itself directly.

1.4 Provide your attack as a self-contained program written in Python

`sol1.py` provided.

1.5 Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

We can use safer alternatives for standard C functions. In this particular example, `strncpy` could be a viable alternative which prevents a buffer overflow attack.

2 vulnerable 2

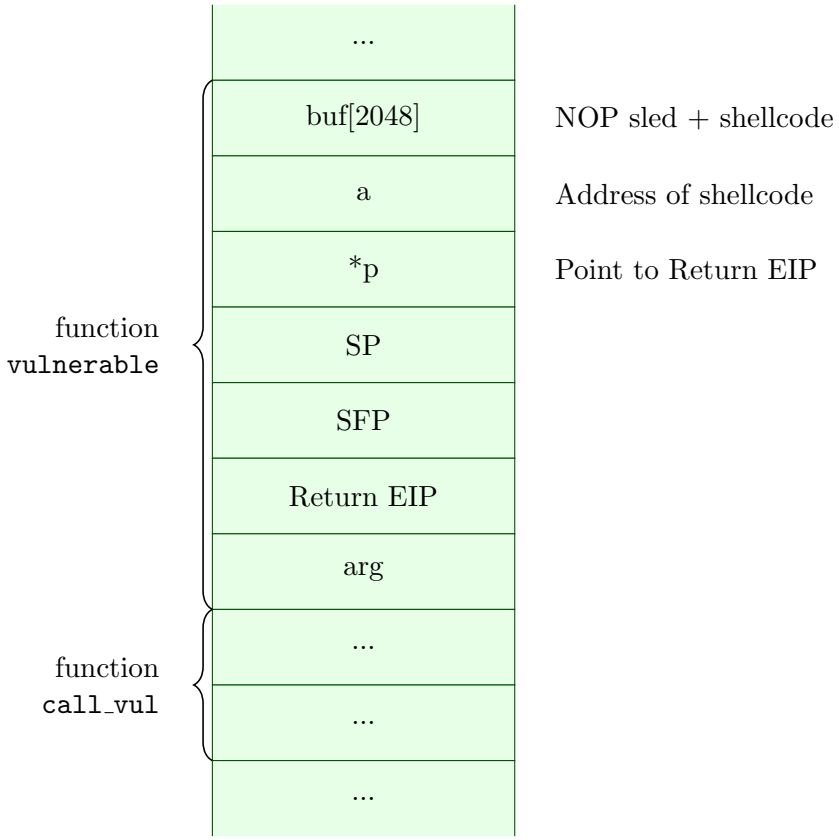
2.1 Briefly describe the behavior of the program

The program takes a command line argument as a "string" input. It then proceeds to copy the same string onto a 2048-byte buffer in memory.

2.2 Identify and describe the vulnerability as well as its implications

The program uses the safer `strncpy` function, but messes up in the limit parameter. It copies over 8 extra bytes onto the buffer, which we can exploit.

2.3 Discuss how your program or script exploits the vulnerability and describe the structure of your attack



In this vulnerability, we fill up the 2048-byte buffer with a NOP sled and the shellcode. For the additional 8bytes that `strncpy` copies over, the values of `a` and `*p` will get altered. We fill in an address in memory which will point to the injected shellcode into `a`. We fill in the address of the memory holding the `Return EIP` into `*p`.

When the program executes the below line, it de-references the memory location holding the stack frame's return EIP value, and replaces it with the address of the shellcode.

```
*p = a;
```

2.4 Provide your attack as a self-contained program written in Python

`sol2.py` provided.

2.5 Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

```
void vulnerable(char *arg){
 int *p;
 int a;
 char buf[2048];
 fprintf(stderr, "%s\n ", buf);
 strncpy(buf, arg, sizeof(buf) + 8);
 *p = a;
}

void vulnerable(char *arg){
 int *p = null; // why is this even here?
 int a = 0; // why is this even here?
 char buf[2048];
 fprintf(stderr, "%s\n ", buf);
 strncpy(buf, arg, sizeof(buf));
 // *p = a; Can't dereference a null pointer
}
```

The original vulnerable function, and the corrected version is placed above.

3 vulnerable 3

3.1 Briefly describe the behavior of the program

This program takes a filename as it's command line input. Then it reads 32-bit chunks from the file as binary data. The first 32-bit chunk is read as the `count`, and then it goes on to read `count` number of further 32-bit data and puts it to a buffer (of size `count`).

3.2 Identify and describe the vulnerability as well as its implications

```
unsigned int count;
fread(&count, sizeof(unsigned int), 1, f);
unsigned int *buf = alloca(count * sizeof(unsigned int));
```

The vulnerability lies in this code segment. The most important thing is probably the fact that `alloca` is being used instead of something like `malloc`. This allocates the buffer inside the stack frame, and not on the heap, and hence we can exploit the stack frame.

As we can see, `count` is an unsigned integer (4bytes). The parameter of `alloca` is also an unsigned int type. If we fill `count` with a large enough value such that the expression `count * sizeof(unsigned int)` leads to an arithmetic overflow, we can probably allocate a small buffer, but end up filling the small buffer with a larger number of elements.

3.3 Discuss how your program or script exploits the vulnerability and describe the structure of your attack

We know that C's `unsigned int` type is of 4 bytes.
It can hold a maximum value of $2^{32} - 1 = 4,294,967,295$

```
unsigned int *buf = alloca(count * sizeof(unsigned int));
```

To get into an arithmetic overflow, we can set the value of count to $\longrightarrow$

$$\frac{4,294,967,295}{4} = 1,073,741,823$$

This will result to the expression returning a result of 0.

From here, we can work our way up and get results of $1, 2, 3$.....*etc.*

This ultimately means that if `count` is set to $1,073,741,885$ for example, the buffer is set to 245 bytes. But $1,073,741,885$ 4-byte chunks will be read and put to the same 245-byte buffer, hence causing an overflow.

$(1,073,741,885 * 4) \mod 4,294,967,295 = 245$

After debugging in GDB and analyzing core dumps, I found that I had to write out 300bytes to the buffer in order to reach the saved EIP. So I constructed an exploit string with 148 bytes of NOPs + 53 bytes shellcode + 99 bytes of more NOPs + 4 byte Return address.

3.4 Provide your attack as a self-contained program written in Python

`sol3.py` provided.

3.5 Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

The error to avoid here is arithmetic/integer overflow. We can either put in more manual checks in our program, or just use larger data types (like `long`) to counter this.

```
// actual code
void read_file(char *name) {
    FILE *f = fopen(name, "rb");
    if (!f) {
        fprintf(stderr, "Error: Cannot open file\n");
        return;
    }
    unsigned int count;
    fread(&count, sizeof(unsigned int), 1, f);
    unsigned int *buf = alloca(count * sizeof(unsigned int));
    if (!buf) {
        return;
    }
    read_elements(f, buf, count);
}

// suggestions for mitigating overflow
void read_file(char *name) {
    FILE *f = fopen(name, "rb");
    if (!f) {
        fprintf(stderr, "Error: Cannot open file\n");
        return;
    }
    unsigned int count;
    fread(&count, sizeof(unsigned int), 1, f);
    if((count * sizeof(unsigned int)) <= count) {
        // integer overflow/zero/negative integer detected
        return;
    }
    unsigned int *buf = alloca(count * sizeof(unsigned int));
    if (!buf) {
        return;
    }
    read_elements(f, buf, count);
}
```

4 vulnerable 4

4.1 Briefly describe the behavior of the program

This is identical to program 1, but with a larger buffer of 1024 bytes. It just copies a command line string argument into the buffer.

4.2 Identify and describe the vulnerability as well as its implications

The string argument is copied onto the buffer using the following line of C code.

```
strcpy(buf, arg);
```

The C library function `char *strcpy(char *dest, const char *src)` copies the string pointed to, by `src` to `dest`. It does not perform any sort of bounds checking, and this is where extra data can get copied to the destination address and might lead to stack/heap corruption.

4.3 Discuss how your program or script exploits the vulnerability and describe the structure of your attack

So the basic concept for this vulnerability was to overflow the 1024 byte wide character buffer, and to extend into the saved EIP for the previous stack frame. The buffer payload is carefully crafted to contain a shellcode, padded on both sides with NOP sleds. The address to place on the "Return EIP" could be anywhere in that NOP sled, or could point to the shellcode itself directly.

However, ASLR was turned on for this binary, and hence the "return address" is not static across program executions. The program helps out a bit in this regard because of a larger buffer size. The shellcode is of 53 bytes, so that leaves us with a lot of space (971 bytes) for the NOP sled.

The ASLR brings a randomization of upto 255 bytes in the address space. So I used an address almost towards the middle of the 1024 byte buffer, and that seemed to work consistently.

4.4 Provide your attack as a self-contained program written in Python

`sol4.py` provided.

4.5 Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

ASLR is a pretty good way of preventing these types of attacks. However, since the buffer is large, it allowed us to do a big NOP spray and guess an offset which probably works consistently across runs. Here if the buffer was smaller, it would have been difficult to guess a return address which works perfectly.

We also need to use safer alternatives for standard C functions. In this particular example, `strncpy` could be a viable alternative which prevents a buffer overflow attack.

5 vulnerable 5

5.1 Briefly describe the behavior of the program

The program takes a filename as a command line input. It reads the file size and allocates a buffer of that size in memory. It then asks for user input in this form

```
(r)ead,[offset] or (w)rite,[offset],[value], (s)ave/quit or (q)uit:
```

If r, the program will read values from the file based on the offset.
If w, the program will write values to a the allocated buffer in memory from the file based on the offset.
If s or q, the program will exit.

5.2 Identify and describe the vulnerability as well as its implications

Since the input size, and the actual input is both fed from the user input, we can misuse that. For example, we can feed the program a small file of 8 bytes as input. That will make the program allocate 8 bytes of file buffer in memory. However, we can use the write command to send more values to the buffer and thereby overflow it.

5.3 Discuss how your program or script exploits the vulnerability and describe the structure of your attack

I devised 2 python scripts `sol5_generate_file.py` and `sol5_command_generator.py` to generate the 2 input files for this problem. I couldn't manage to break the program in time unfortunately.

5.4 Provide your attack as a self-contained program written in Python

5.5 Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

We can pass in another argument containing the filesize into the `user_interaction` function, and then place constraint checks on the read and write operation so that the program does't operate on out-of-bounds file positions.

```
int user_interaction(long long filesize, unsigned char * file_buffer) {
    long long offset;
    int value;
    do {
```

```
        get_user_request();
        if (request_buffer[0] == 'r') {
            if (sscanf(request_buffer, "r,%lld\n", &offset) == 1 &&
            offset < filesize)
                printf("%#2.2d\n", file_buffer[offset]);
            else
                puts("error: unrecognized option");
        }
        else if (request_buffer[0] == 'w') {
            if (sscanf(request_buffer, "w,%lld,%d\n", &offset, &value) == 2 &&
            offset < filesize)
                file_buffer[offset] = value;
            else
                puts("error: unrecognized option");
        }
        else if (request_buffer[0] == 'q' || request_buffer[0] == 's') {
            puts("exiting application");
        }
        else {
            puts("error: unrecognized option");
        }
    } while (request_buffer[0] != 'q' && request_buffer[0] != 's');
    return (request_buffer[0] == 'q')? QUIT : SAVE_QUIT;
}
```