Buffer Overflow Background

EN.605.611: Foundations of Computer Architecture
Chuck Norris

Introduction

A buffer is a span of memory that is "reserved" for some set of data. Reserved here is in quotations because at lower levels the reservation of space is typically by convention and contract only. That is to say a compiled C program will respect conventions but one can assemble their own program which fails to follow them. Flawed implementations of this contract or poor application-programmer interfaces can lead to vulnerabilities such as a buffer overflow.

At a high level a buffer overflow is when a span of memory on the stack is allocated for a certain type of data. When data is written to that span of memory it exceeds the allocation size and overflows into adjacent memory spaces. This can lead to data corruption and even to hijacking execution of the program. There are ways to mitigate buffer overflows such as stack canaries. First though we need to understand how data is places into memory.

The Stack

The stack is a data structure you may have learned about in a previous course. It's a variable length structure and in this context you can think of it as growing upward toward lower addresses. With rare exceptions most modern systems have stacks oriented in this way. In practice, a simplified version might look something like this:

```
        Address     Data
        ...
Top     0x00004124  ????
        0x00004128  ????
        ...
        ...
        0x00008FF8  ????
Bottom  0x00008FFC  ????
```

Within a program the stack is used to keep track of local variables and general data that processes and functions need to execute in a well-behaved manner. When you call a function some space is allocated on the stack by subtracting some amount from the stack pointer– adding free space to the top of the stack. The stack pointer is stored in a special register, SP or R13, to keep track of the top of the stack. Data can also be pushed onto the stack. Pushing and popping instructions increments and decrements the stack pointer respectively. When a function exits the same amount is added back to the stack pointer, or pop instructions are used, to effectively "remove" that space. The data added by the function is referred to as the function's stack frame.

```
/* Stack Frame */
      +-----------------+ <-- low address
      |  Top of Stack   |
      +-----------------+
      | Local Variables |
      +-----------------+
      |  Frame Pointer  |
      +-----------------+
      | Return Address  |
      +-----------------+
      |   Rest of the    |
      |      Stack       |
      +------------------+ <-- high address
```

Stacks typically "grow" toward lower address values. This isn't necessary but is true in most modern systems. Lets take a look at an example of a stack before and after the main function of an arm program is called.

```
<main>:
  e52db004     push     {fp}       ; Push fp onto the stack, adjust sp.
  e28db000     add      fp, sp, #0
  e24dd01c     sub      sp, sp, #8 ; Add 8 bytes of space to stack.
```

```
  /* Stack Before: */
         Address     Data
         ...
         0x00004118  ????
         0x0000411C  ????
         0x00004120  ????
 SP ---> 0x00004124  ????   // Current Stack Pointer; Top of Stack
         0x00004128  ????
         ...
         ...
         0x00008FF8  ????
         0x00008FFC  ????   // Bottom of the Stack

    /* Stack After */
         Address     Data
         ...
 SP ---> 0x00004118  ????   // New Stack Pointer; Top of Stack
         0x0000411C  ????
         0x00004120  fp
         0x00004124  ????   // Old Stack Pointer
         0x00004128  ????
         ...
         ...
         0x00008FF8  ????
         0x00008FFC  ????   // Bottom of the Stack
```

Some function calling conventions, such as x86, arguments are pushed onto the stack prior to the function call. After the function is called space is made, by subtraction a value from the stack pointer, for local variables. In ARM the function calling convention only uses stack space if the there are more than 3 arguments.

Suppose we have a very simple C function:

```
int my_func(int A) { // A = 0xAA
    int C = 0xCC;
    int B = 0xBB;
    return B;
}
```

Before the program enters this function the argument A will be stored in R0. Then when the function is entered 4 bytes, or 8 if 64 bit, will be subtracted from the stack pointer for each local variable. Then the value(s) is moved onto the stack. The stack will look something like:

```
        Address     Data
        ...
        0x00004118  ????
SP ---> 0x0000411C  0x00BB   // int B "pushed" onto the Stack second
        0x00004120  0x00CC   // Int C "pushed" onto the Stack first
        0x00004124  ????
        0x00004128  ????
        ...
        ...
```

A Buffer Overflow

If *int C* in *my_func* from the previous section was instead an eight byte character array, the space for this array would be on the stack.

```
int my_func1(int A) { // A = 0xAA
    int B = 0xBB;
    char C[8];
    return B;
}
```

```
        Address    Data
        ...
SP ---> 0x00004124 .. .. .. .. // Space for eight
        0x00004128 .. .. .. .. // bytes of the char array
        0x0000412C BB 00 00 00 // First local variable
        ...
```

When writing data to a buffer, and in general, the first byte is written at address X and the next byte at X+1. In the above example the twelfth byte will be at address 0x0000412B. The address just before the space for int *B*. If a function or logic were present that implemented the space reservation contracts incorrectly then we can write past 0x0000412B and alter the data contained in the space for int *B*.

The C function *strcpy* takes a string and copies it to another part of memory. It takes two arguments: destination and source. There are no bounds checks on the write. That is to say it will copy the contents of the source to the destination, byte by byte, until it comes across a null character used to terminate the source string. The user of *strcpy* is expected to check that the string contained in the source will fit in the destination buffer. If it doesn't *strcpy* is content to continue writing bytes until it either reaches the end of the string or causes an error (segmentation fault).

Using *strcpy* we can create a *my_func2* that will overwrite the local variable B.

```
  int my_func2(char *A) {
  int B = 0xBB;
  char C[8];

    // Stack before strcpy
  strcpy(C, A);
    //  Stack after strcpy

  return B;
}
```

Suppose A = "AAAAAAAAA" (9 bytes). This is larger than the space allocated for C by 1 bytes– it doesn't fit.

```
  /* Stack before strcpy */
    Address    Data
    0x00004124 .. .. .. ..
    0x00004128 .. .. .. ..
    0x0000412c BB 00 00 00
      ........ .. .. .. ..

Then strcpy moves A into C
  /* Stack after strcpy */
    Address    Data
    ...        .. .. .. ..
    0x00004124 41 41 41 41 // 0x41 = 'A'
    0x00004128 41 41 41 41
    0x0000412c 41 00 00 00
      ........ .. .. .. ..
```

Strcpy have overwritten the value in int B giving it a new value of 0x41! We can do the same with 10 bytes or more. There is a limitation associated with *strcpy* when trying to exploit a buffer overflow vulnerability. The buffer can only be written with printable ascii characters. This may sometimes be enough though.

```
  int my_func3(char *A) {
    int B = 0x00;
    char C[8];

    strcpy(C, A);

    if B != 0 {
      inaccessible_function();
    }

    return B;
  }
```

At a glance it appears as though *inaccessible_function()* should never be executed. This is because we initialize B to zero, never again write to it, then test if B is not zero. However, as we showed above, we can overwrite B with a non-zero value. This is probably not what the programmer would have intended but due to the ordering of the local variables on the stack and the vulnerable *strcpy* we can alter the value of B and get access to the *inaccessible_function*.

A mitigation for this would be to change the order of the local variables. This would place B above C on the stack. Then when C overflows down the stack, B is out of the vulnerable stack space. With this change, the *inaccessible_function* would be inaccessible.

Local variables are not the only thing stored on the stack– recall the functions stack frame. When you enter a non-leaf function the address of the next instruction is pushed onto the stack, before local variables, to be saved. A leaf function is one that does not call any other functions. Then the function exits it pops the address off the stack into the lr register (ARM) or eip/rip (x86/64).

```
  /* Stack of my_func3 after pushing local variables to stack */
    Address    Data
```

```
0x00004124 00 00 00 00
0x00004128 .. .. .. ..
0x0000412C .. .. .. ..
0x00004130 48 93 7f 00 // lr value prior entering my_func3
.......... .. .. .. ..
// Rest of stack
```

If this value is corrupted during function execution it will get popped off the stack and the CPU will try to execute the the corrupted value. Should the corrupted value be a valid instruction address the CPU will execute it however if it is not a valid address it will crash the program. This could allow us to control what code the CPU executes!

Unfortunately, we cant mitigate this by reordering any local variables. However, if we expect their to be an overflow we could place a special value, one we know before hand, on the stack between the lr register and the local variables we can test it before the function exits. If the value isn't what we expected we can determine there was a stack overflow and prevent the function from returning by throwing an error. This will stop any overflow from executing unauthorized instructions. This special value is called a Stack Canary or Stack Cookie.

Stack Canaries

A stack canary is a filler word or data placed between function data pushed onto the stack and other data in the stack frame. This filler word can be random, static, or hybrid. A random stack canary will provide greater security as there is little chance for an attacker to know the value prior to execution of the program.

```
/* Stack without Canary */
  +--------------------+ <---- Top of the stack
  |       Buffer       |
  |                    |
  |                    |
  +--------------------+
  |  Local Variables   |
  +--------------------+
  |   Frame Pointer    |   // fp
  +--------------------+
  |  Return Address    |   // lr
  +--------------------+
  |    Rest of the     |
  |       Stack        |
  +--------------------+ <---- Bottom of the stack

/* Stack with Canary */
  +--------------------+ <---- Top of the stack
  |       Buffer       |
  |                    |
  |                    |
  +--------------------+
  |  Local Variables   |
  +--------------------+
  |   Stack Canary     | <---- Canary between buffer and Return Address
  +--------------------+
  |   Frame Pointer    |   // fp
  +--------------------+
  |  Return Address   |   // lr
  +-------------------+
  |    Rest of the    |
  |       Stack       |
  +-------------------+ <---- Bottom of the stack
```

When a function is entered, the stack canary will be pushed onto the stack. After this the function continues to execute as it normally would. At the end of the function, prior to returning, the value of the canary is checked for integrity. If they value is corrupted the an exception is thrown. The creation of the canary and the checking of its values are done via function calls.

In order to maximize the security offered by stack canaries they should be employed in every function and use random values only known at run-time. These two things can be set with compiler flags and most modern systems have these enabled by default.

References

[One96] Aleph One. “Smashing the Stack for Fun and Profit”. In: *.::Phrack Magazine::.* 49.14 (1996).

[Eri08] Jon Erickson. *Hacking: The Art of Exploitation, 2nd Edition.* No Starch Press, 2008. ISBN: 1593271441.

[0xr] 0xricksanchez. *Exploit Mitigation Techniques - Stack Canaries.* URL: `https://0x00sec.org/t/exploit-mitigation-techniques-stack-canaries/5085`. (accessed: 04.10.2021).