CSCI 331: Introduction to Computer Security

Lecture 14: Stack Smashing

Instructor: Dan Barowy

Williams

Topics

Buffer overflow exploit: overview

Using GDB to analyze a program

Crafting inputs

Your to-dos

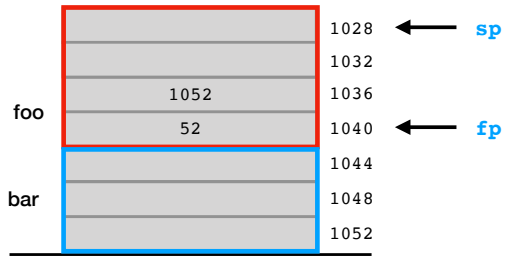
1. Reading response (Miller), **due Wed 11/3**.
2. Lab 5, **due Sunday 11/7**.
3. Project part 2, **due Sunday 11/14**.

Quiz

Buffer overflow exploits

Recall: preamble stores retaddr

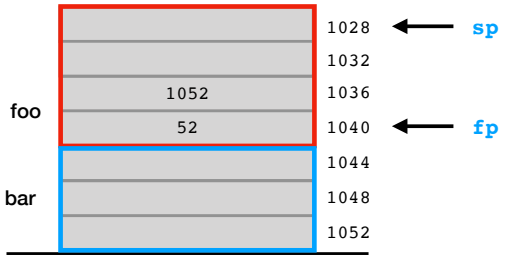
- The callee's preamble stores the retaddr to the stack.
- E.g., foo stores the retaddr for bar at the location pointed to by fp.
- foo does this so that if it calls another function (e.g., printf), which would overwrite the retaddr in lr, it can just restore it from the stack.
- The epilogue restores the retaddr from the stack and then jumps to that address.
- E.g.,
  sub sp, fp, #4
  pop {fp, pc}



Recall: preamble stores retaddr

- The callee's preamble stores the retaddr to the stack.
- E.g., foo stores the retaddr for bar at the location pointed to by fp.
- foo does this so that if it calls another function (e.g., printf), which would overwrite the retaddr in lr, it can just restore it from the stack.
- The epilogue restores the retaddr from the stack and then jumps to that address.
- E.g.,

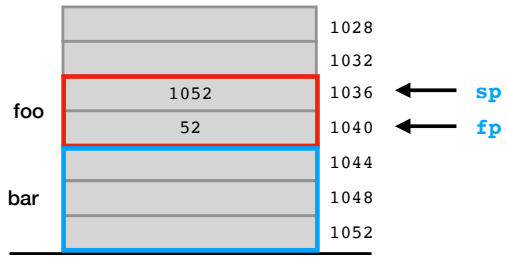
pc → sub sp, fp, #4
pop {fp, pc}



Recall: preamble stores retaddr

- The callee's preamble stores the retaddr to the stack.
- E.g., foo stores the retaddr for bar at the location pointed to by fp.
- foo does this so that if it calls another function (e.g., printf), which would overwrite the retaddr in lr, it can just restore it from the stack.
- The epilogue restores the retaddr from the stack and then jumps to that address.
- E.g.,
  sub sp, fp, #4

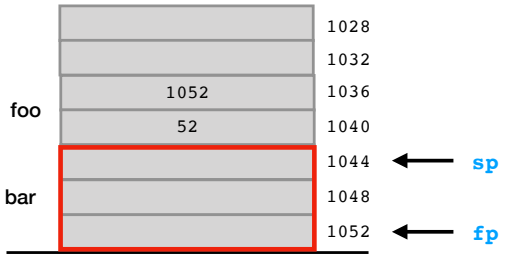
pc → pop {fp, pc}



Recall: preamble stores retaddr

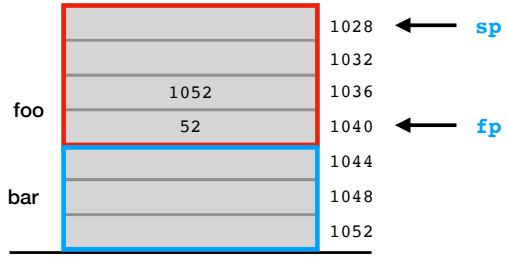
- The callee's preamble stores the retaddr to the stack.
- E.g., foo stores the retaddr for bar at the location pointed to by fp.
- foo does this so that if it calls another function (e.g., printf), which would overwrite the retaddr in lr, it can just restore it from the stack.
- The epilogue restores the retaddr from the stack and then jumps to that address.
- E.g.,
  sub sp, fp, #4
  pop {fp, pc}

`pc` = 52



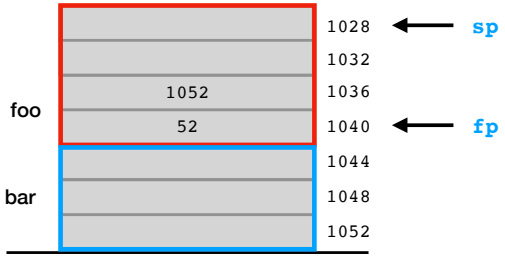
Buffer overflow exploit

- The goal of a buffer overflow exploit is to overflow a buffer such that we also corrupt the return address.
- Suppose an 8-byte buffer starts at 1028.
- If we write >8 bytes, values overflow into the parts of the stack that store control values.
- E.g., suppose we want to jump to a completely different function that happens to be at address 192.



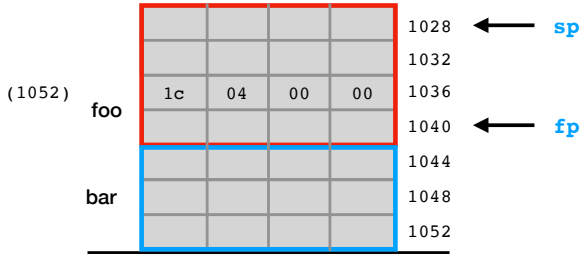
Buffer overflow exploit

- It helps to see values at the byte level.
- We'll also use hexadecimal.
- 1052 = 0 x 00 00 04 1c
- 52 = 0x 00 00 00 34
- Remember that ARM is little-endian, so the little end is stored first.



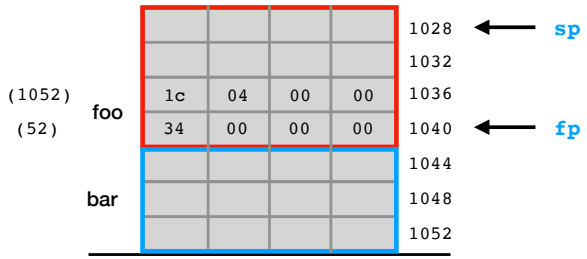
Buffer overflow exploit

- It helps to see values at the byte level.
- We'll also use hexadecimal.
- 1025 = 0 x 00 00 04 1c
- 52 = 0x 00 00 00 34
- Remember that ARM is little-endian, so the little end is stored first.



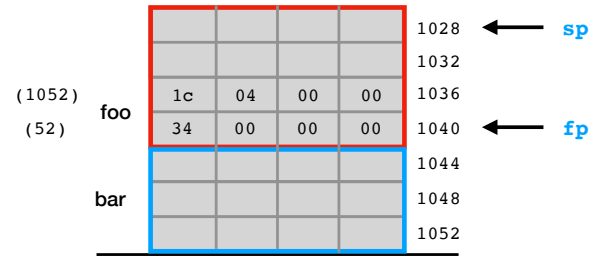
Buffer overflow exploit

- It helps to see values at the byte level.
- We'll also use hexadecimal.
- 1025 = 0 x 00 00 04 1c
- 52 = 0x 00 00 00 34
- Remember that ARM is little-endian, so the little end is stored first.



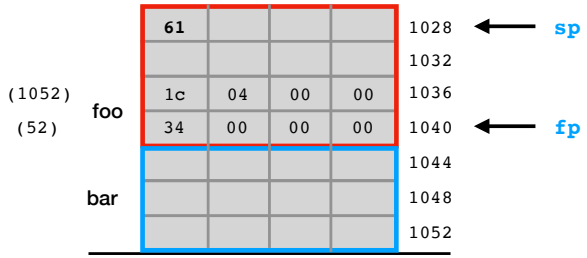
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about "abcdefghijkl\xc0\x00\x00\x00"?



Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about "abcdefghijkl\xc0\x00\x00\x00"?

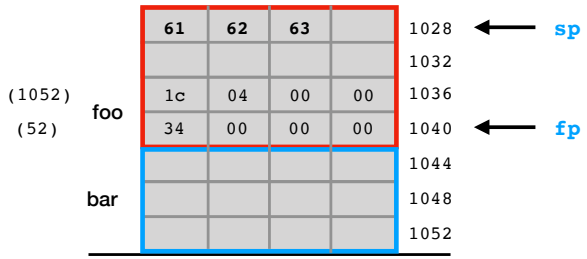


Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about "abcdefghijkl\xc0\x00\x00\x00"?

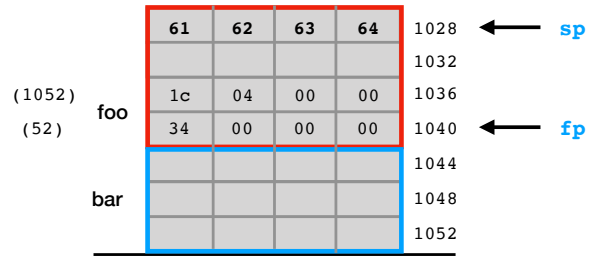
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



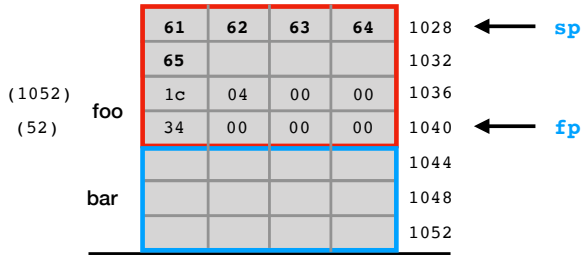
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



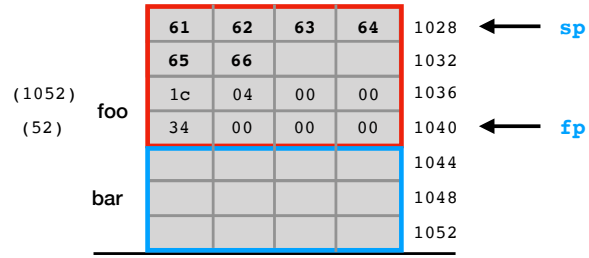
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



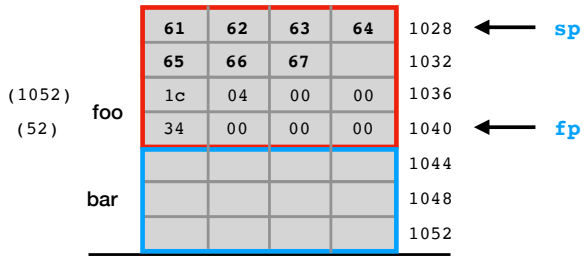
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



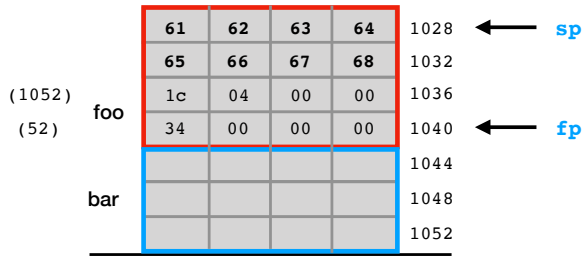
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



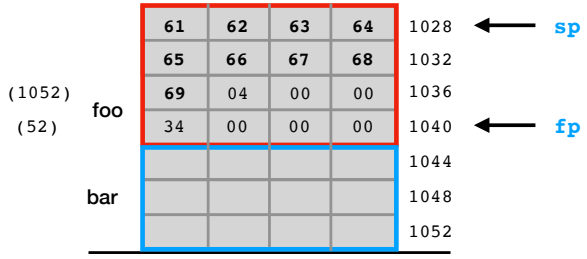
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



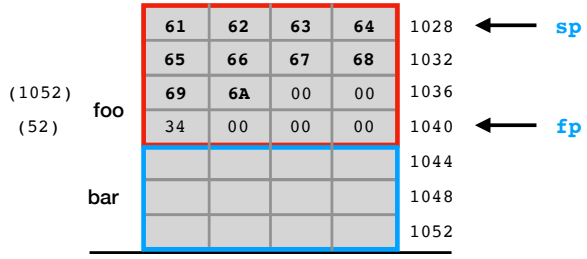
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



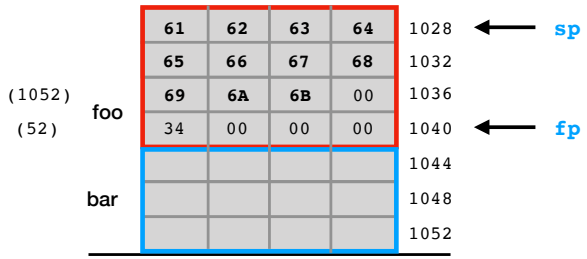
Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?

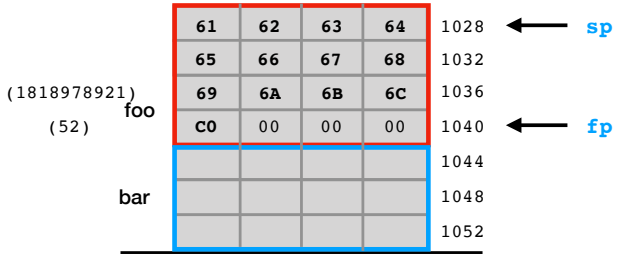


Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?

Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?

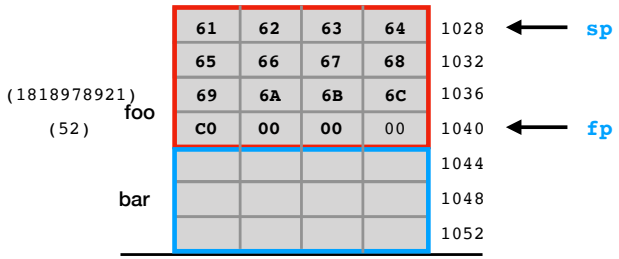


Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?

Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?



Buffer overflow exploit

- 192 = 0x 00 00 00 c0
- We just need to write 16 bytes, ending with 0x000000c0.
- How about “abcdefghijkl\xc0\x00\x00\x00”?

Buffer overflow exploit

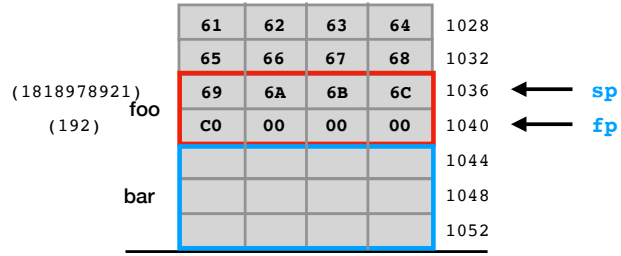
- Now when foo returns, it returns to the wrong place.

pc → sub sp, fp, #4
pop{fp, pc}

Buffer overflow exploit

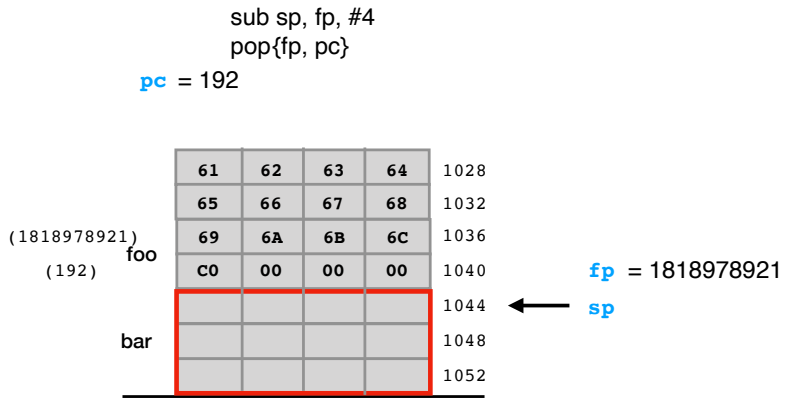
- Now when foo returns, it returns to the wrong place.

sub sp, fp, #4
pc → pop{fp, pc}



Buffer overflow exploit

- Now when foo returns, it returns to the wrong place.



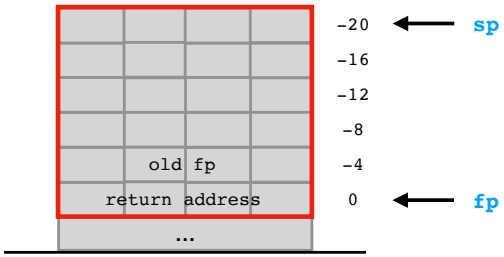
Crafting inputs

globalthermonuclearwar.c

Remember this program?

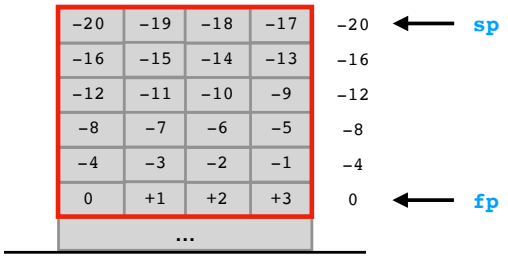
globalthermonuclearwar.c

authenticate_and_launch function



globalthermonuclearwar.c

authenticate_and_launch function



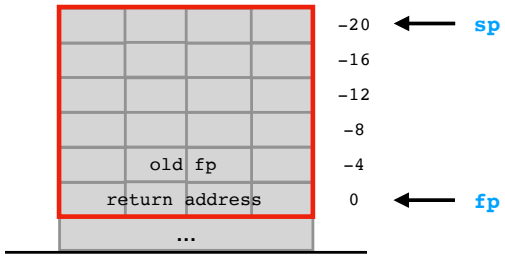
Using GDB to find locations of locals

globalthermonuclearwar.c

```
 1 void    authenticate_and_launch(void) {
 2   int n_missiles = 2;
 3   bool allowaccess = false;
 4   char response[8];
 5
 6   printf("Secret: ");
 7   gets(response);
 8
 9   if (strcmp(response, "Joshua") == 0)
10     allowaccess = true;
11
12   if (allowaccess) {
13     puts("Access granted");
14     launch_missiles(n_missiles);
15   }
16
17   if (!allowaccess)
18     puts("Access denied");
19 }
```

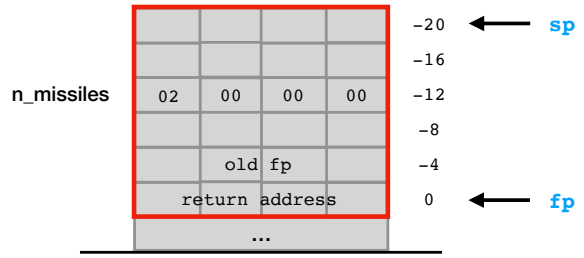
globalthermonuclearwar.c

authenticate_and_launch function



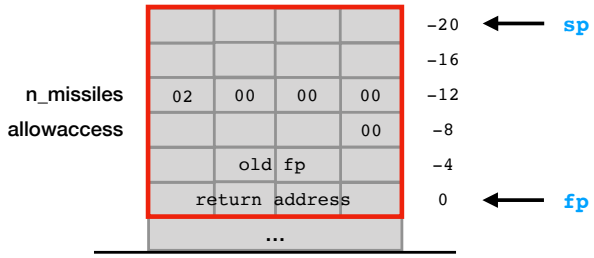
globalthermonuclearwar.c

authenticate_and_launch function



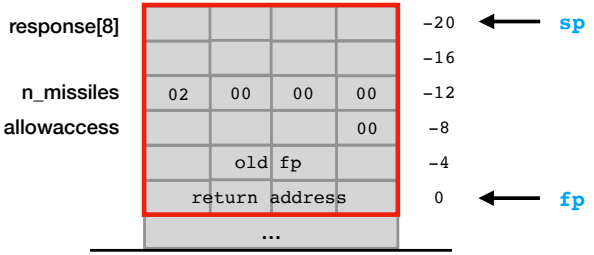
globalthermonuclearwar.c

authenticate_and_launch function



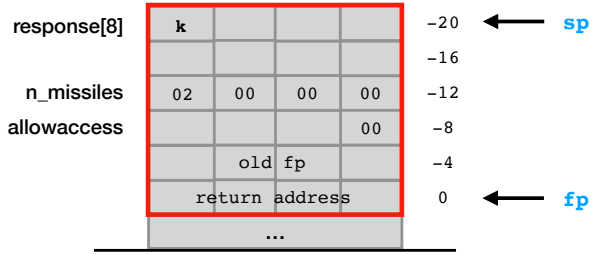
globalthermonuclearwar.c

authenticate_and_launch function



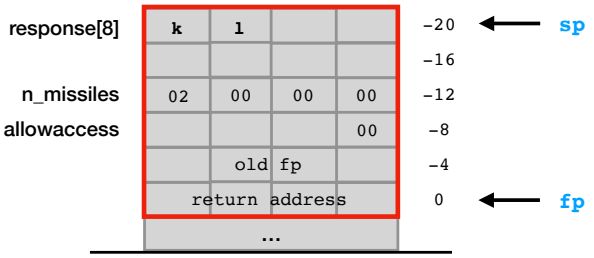
globalthermonuclearwar.c

authenticate_and_launch function



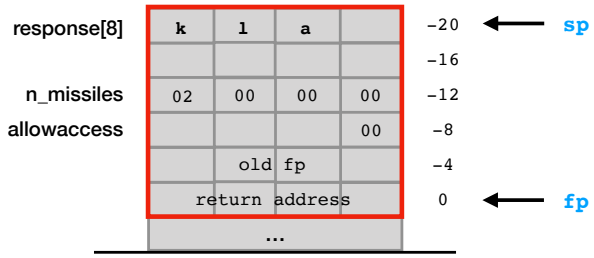
globalthermonuclearwar.c

authenticate_and_launch function



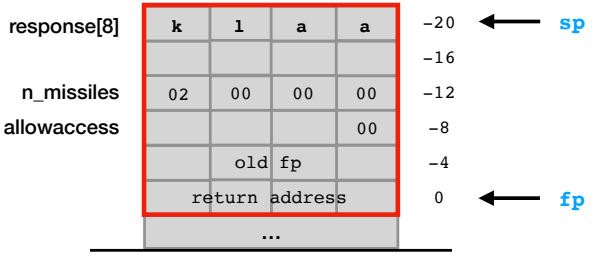
globalthermonuclearwar.c

authenticate_and_launch function



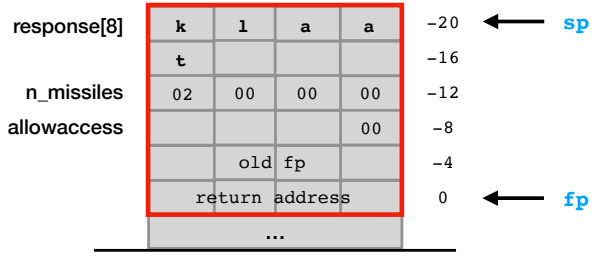
globalthermonuclearwar.c

authenticate_and_launch function



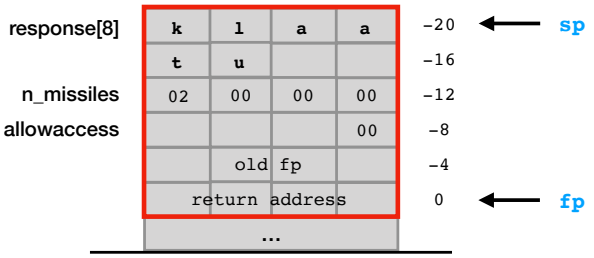
globalthermonuclearwar.c

authenticate_and_launch function



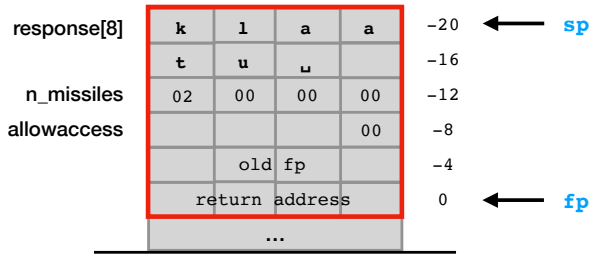
globalthermonuclearwar.c

authenticate_and_launch function



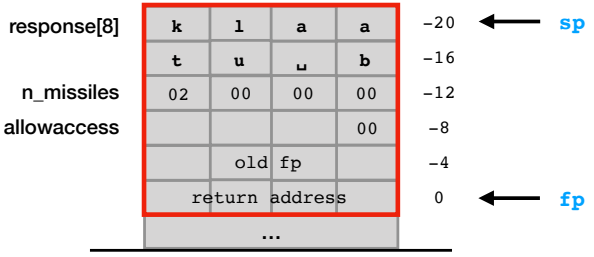
globalthermonuclearwar.c

authenticate_and_launch function



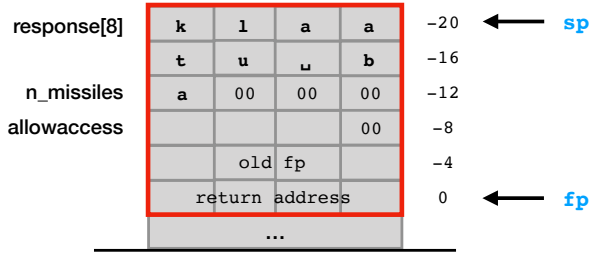
globalthermonuclearwar.c

authenticate_and_launch function



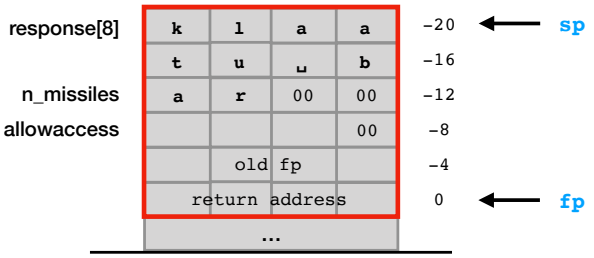
globalthermonuclearwar.c

authenticate_and_launch function



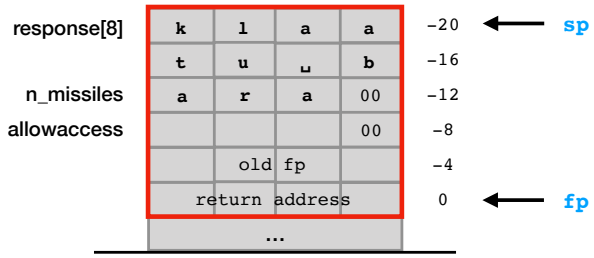
globalthermonuclearwar.c

authenticate_and_launch function



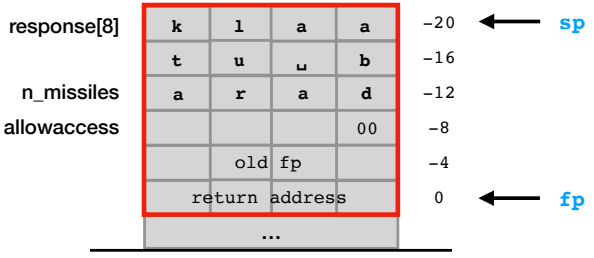
globalthermonuclearwar.c

authenticate_and_launch function



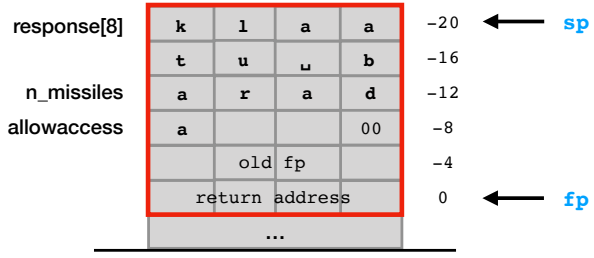
globalthermonuclearwar.c

authenticate_and_launch function



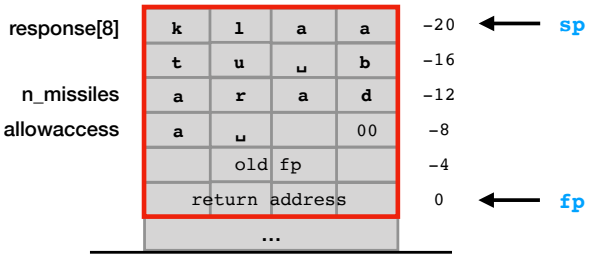
globalthermonuclearwar.c

authenticate_and_launch function



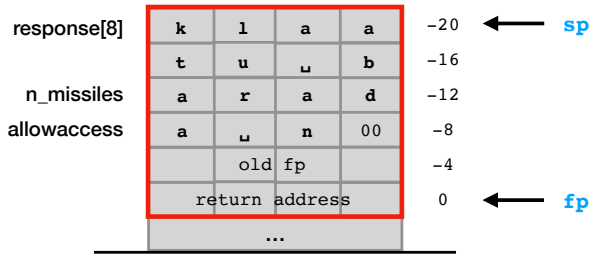
globalthermonuclearwar.c

authenticate_and_launch function



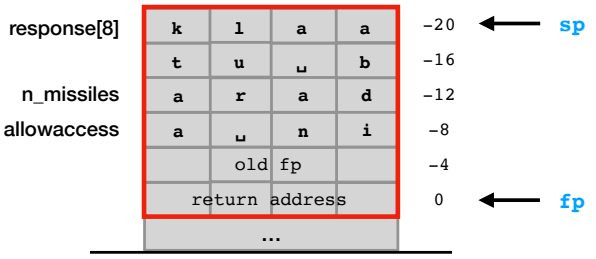
globalthermonuclearwar.c

authenticate_and_launch function



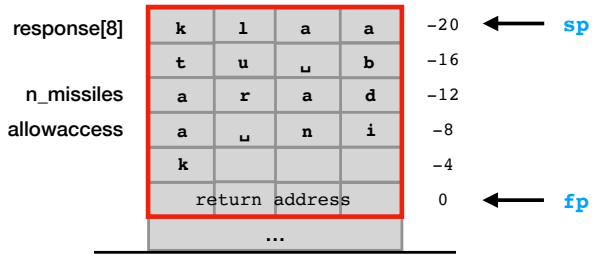
globalthermonuclearwar.c

authenticate_and_launch function



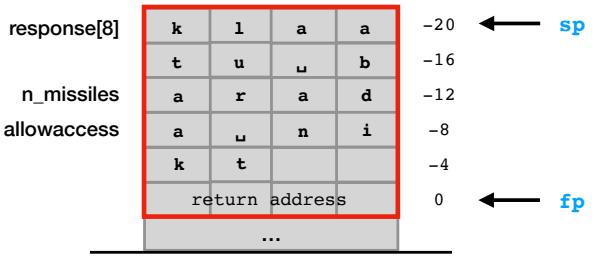
globalthermonuclearwar.c

authenticate_and_launch function



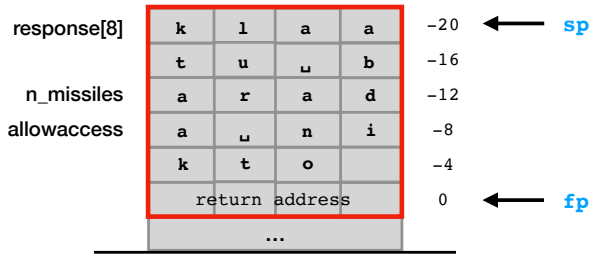
globalthermonuclearwar.c

authenticate_and_launch function



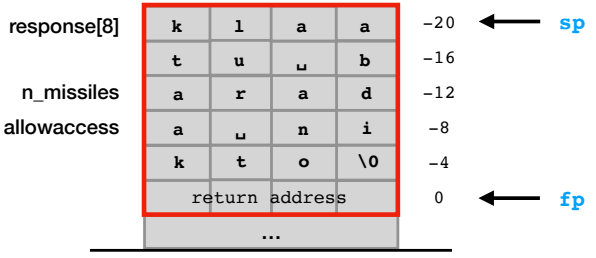
globalthermonuclearwar.c

authenticate_and_launch function



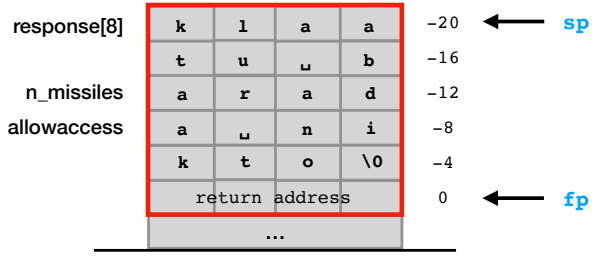
globalthermonuclearwar.c

authenticate_and_launch function



globalthermonuclearwar.c

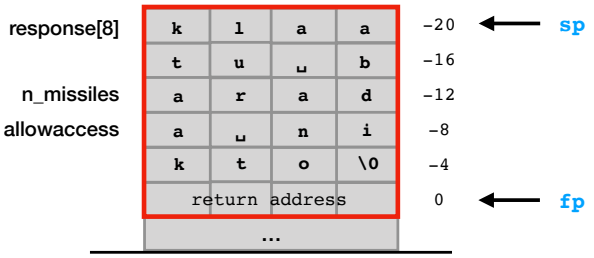
authenticate_and_launch function



What **value** is `allowaccess`? `0x69 >0` → `true`

globalthermonuclearwar.c

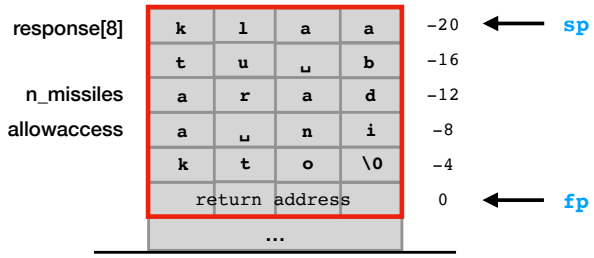
authenticate_and_launch function



What **number** is `n_missiles`?

globalthermonuclearwar.c

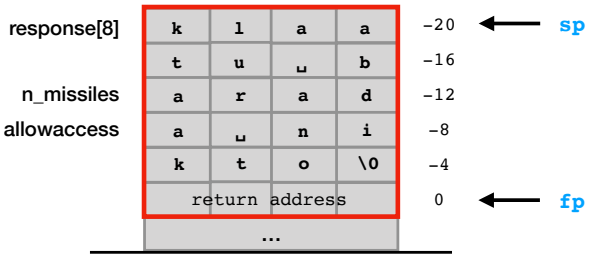
authenticate_and_launch function



What **number** is `n_missiles`? `0x64617261 = 1684107873`

globalthermonuclearwar.c

If I wanted the program to jump to `launch_missiles` by overwriting the return address, what kind of input would I need to give it?



Assume the address of `launch_missiles` is 0x10498.

Recap & Next Class

Today we learned:

Crafting inputs

Next class:

Commonly vulnerable C functions

Stack smashing with shellcode