

CSE 127 Week 3

Discussion

Fall 2026

Reminder

- Due: Thursday, October 22, 2026 by 11:59 PM PDT
- Make sure your cookie is correct
- The score on Gradescope is not your score for the assignment
- Target 0 / 1 / 2 / 3 / 4+ demo - Refer last week's Discussion

Agenda

- Memory Defense Mechanisms Review
 - targets 5, 6, and 7

Stack Canary (Cookie)

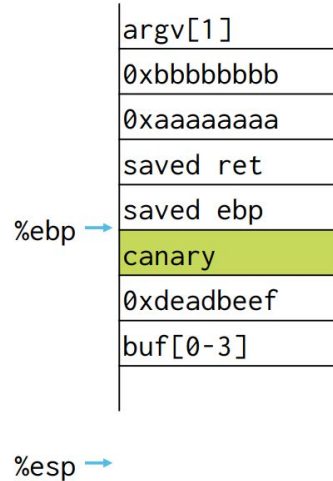
Example (at a high level)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void foo() {
    printf("hello all!!\n");
    exit(0);
}

void func(int a, int b, char *str) {
    int c = 0xdeadbeef;
    char buf[4];
    strcpy(buf, str);
}

int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbb, argv[1]);
    return 0;
}
```

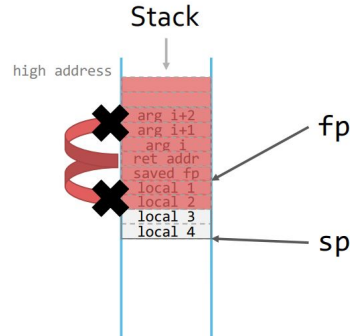


DEP (Data Execution Prevention) / NX / W^XX

Nonexecutable Stack

Mark stack pages as
"nonexecutable"

Attempts to execute
instructions from the stack
trigger memory access
violations



Pros:

- No changes to application software
- Little/no performance impact

Cons:

- Requires hardware support (MPU, MMU, or SMMU)
 - I.e., may not be doable on microcontrollers or some lower-end embedded processors
- Doesn't work automatically with certain programming "tricks"
 - E.g., self-modifying code, compile-in-place/just in time compilation

Target 5 (Hint): Bypassing DEP

- Remember - Compiled with DEP enabled
- **W^X (Write XOR eXecute)!**
- You cannot put shellcode in buffer and point to it from return address.
- What can you do?
- **"Return-to-libc" style of attack**
- What does `system("/bin/sh");` do?

// Compiled with DEP enabled.

```
void greetings(void)
{
    system("echo Hello World");
}

void vulnerable(char *arg)
{
    char buf[10];
    strcpy(buf, arg);
}

int _main(int argc, char *argv[])
{
    ....
    setuid(0);
    vulnerable(argv[1]);
    greetings();
    ....
}
```

Target 5 (Hint): Bypassing DEP

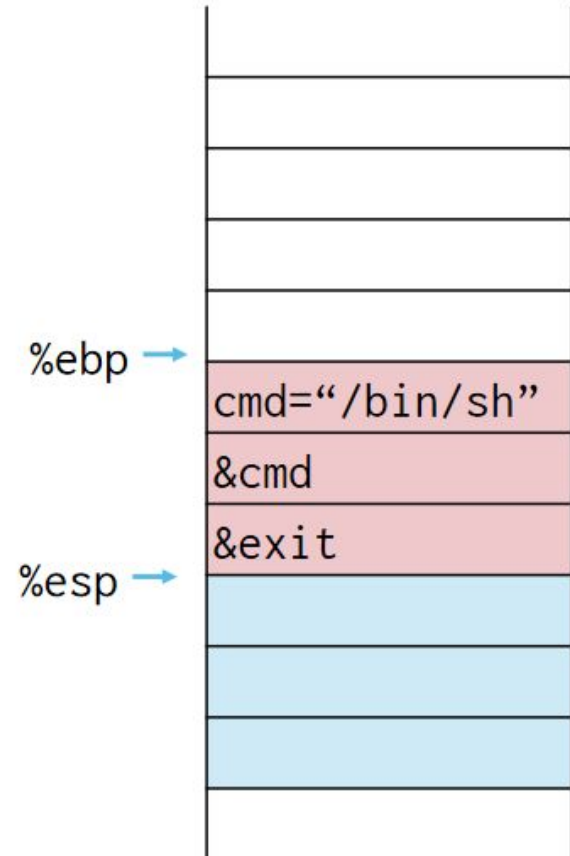
Return to Libc review

- Code Reuse Attack. (Use the code that's already there: Program + shared libraries)
- Need to find the address of:
 - `system()`
 - String `"/bin/sh"`
- Overwrite the return address to point to start of `system()`
- Place address of `"/bin/sh"` on the stack so that `system()` uses it as the argument
 - To be clean, you also want to push `exit()` on the stack so it will shut down gracefully
 - But in this target you don't have to do that. (okay if segfault when root shell exit)

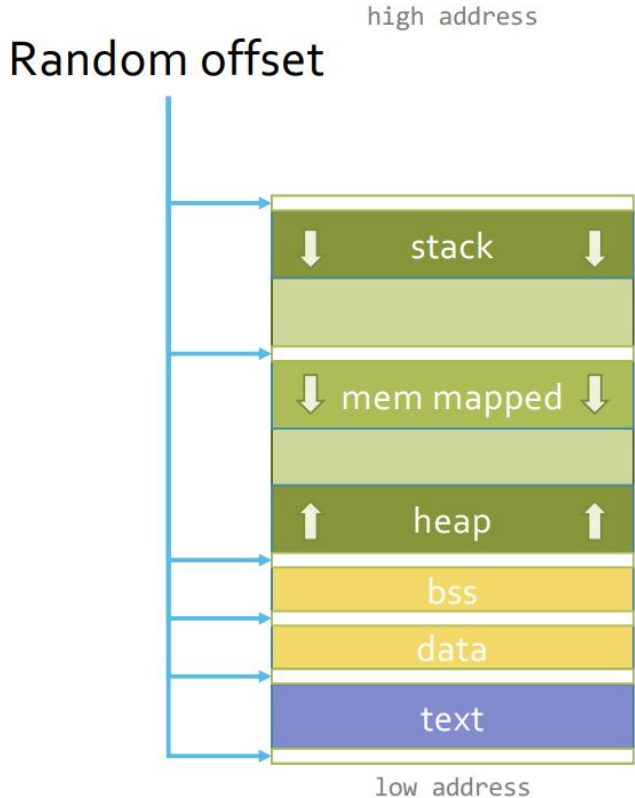
Target 5 (Hint): Bypassing DEP

Return to Libc review

- What we want to get to:
- Transfer control to address of `system()` in libc
- Setup stack frame to look like a normal call to `system()`
 - `int system(const char *command);`
 - `&exit()` system call is in the slot where the return address would be
 - `&cmd` is the argument
 - It points to the string `"/bin/sh"` stored further down the stack



ASLR (Address Space Layout Randomization)



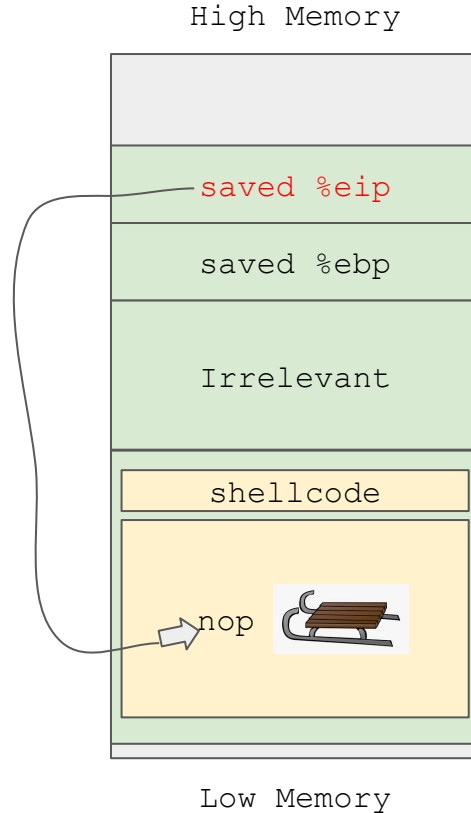
```
tshneour@fedora:~$ cat /proc/sys/kernel/randomize_va_space  
2
```

- 0 - no randomization
- 1 - partial randomization
- 2 - full randomization*

Initial locations of stack, shared libraries, vDSO, and heap* randomized

Target 6 (Hint): Variable stack position

- ASLR (address-space layout randomization) enabled
- Stack can be placed anywhere in memory (0–255 bytes)
- How can I make sure my shell code always hit?
- Hint: NOP sleds
- NOP = 0x90
- If number large enough, there's a region on stack that always have nop in it.



// Compiled with -DMINIASLR.

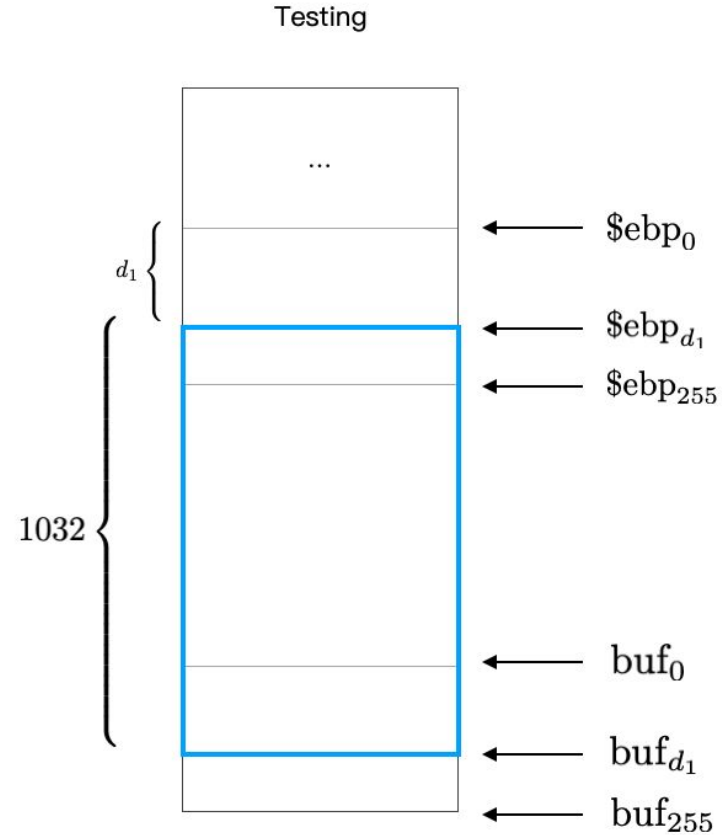
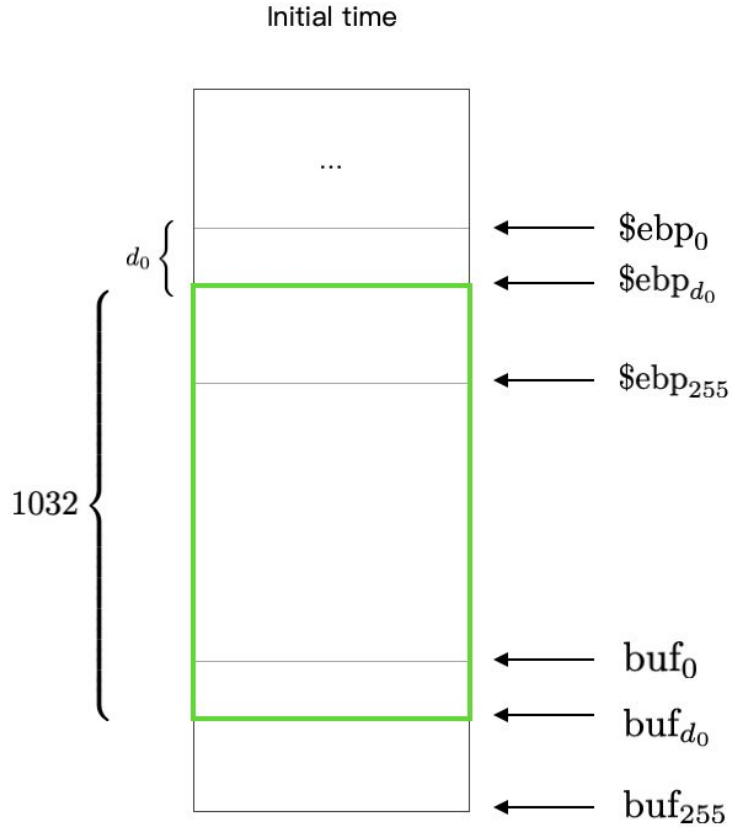
```
void vulnerable(char *arg)
{
    char buf[1024];
    strcpy(buf, arg);
}

int _main(int argc, char *argv[])
{
    ....

    vulnerable(argv[1]);

    return 0;
}
```

Target 6 (Hint): Variable stack position



Target 7: ROP

- Return Oriented Programming
- Extra credit (4 points)
- Identical to target2, but it is compiled with DEP enabled
- Implement a ROP-based attack to bypass DEP and open a root shell.
- **ROPgadget.**
- The --binary, --badbytes, --multibr, and --ropchain flags will be particularly helpful.
- Closest to real-world attacks

```
void vulnerable(char *arg)
{
    char buf[100];
    strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
    if (argc != 2) {
        fprintf(stderr, "Error: need a command-line
argument\n");
        return 1;
    }
    vulnerable(argv[1]);
    return 0;
}
```

```
setuid(0);           //eax = 23
execve("/bin/sh", 0, 0); //eax = 11
```

Memory Defenses: Summary

Defense	Mitigates	Why it helps	Limitation
Canary	Contiguous writes to the stack	Checks a guard value before return	Leaks or writes that skip the guard
DEP / NX	Injected shellcode	Stack pages cannot execute	Return-to-libc / ROP reuse code (T5, T7)
ASLR + PIE	Predictable target addresses	Randomizes memory locations	Leaks defeat secrecy. NOP sleds tolerate small shifts (T6)
Control-Flow Integrity (CFI)	Illegal jumps and returns	Prevents ROP attacks	Hard to strictly enforce without overhead