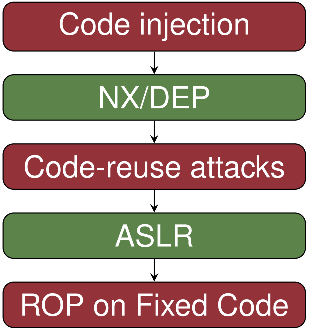
Lec 10: Canary

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Canary

Attack/Defense So Far ...


Code injection

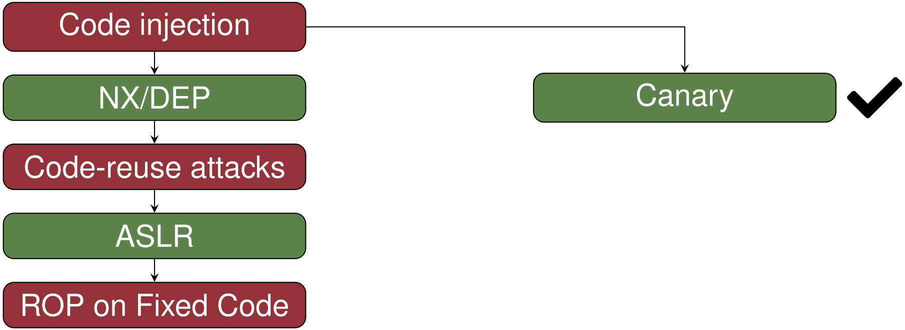
NX/DEP

Code-reuse attacks

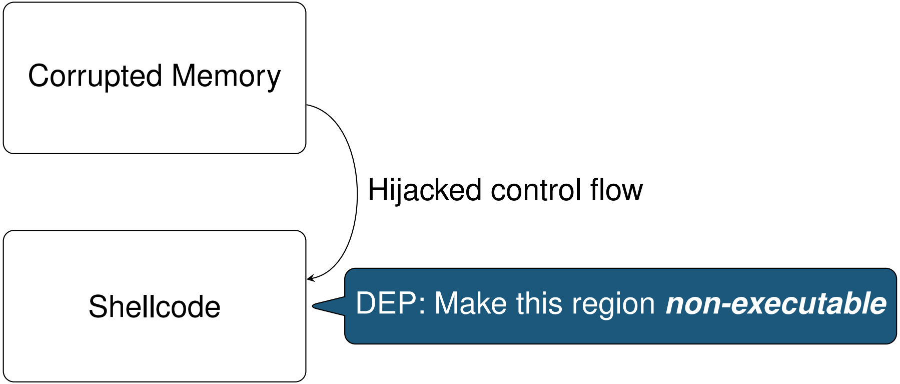
ASLR

ROP on Fixed Code

Attack/Defense So Far ...



Recap: DEP and ASLR



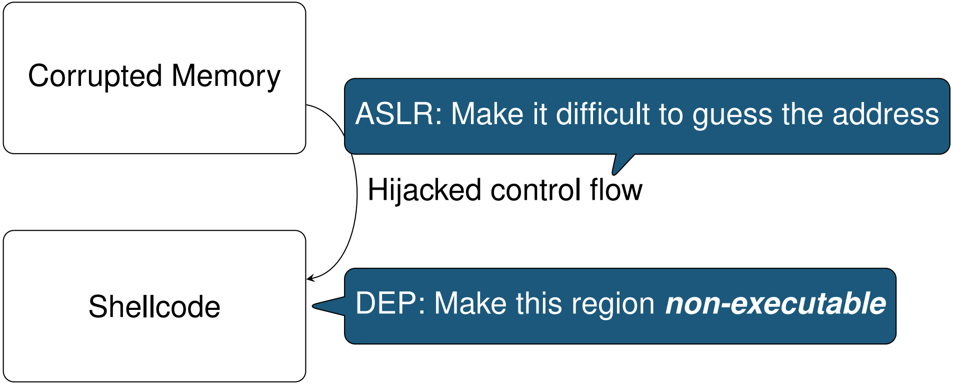
Corrupted Memory

Hijacked control flow

Shellcode

DEP: Make this region ***non-executable***

Recap: DEP and ASLR



Corrupted Memory

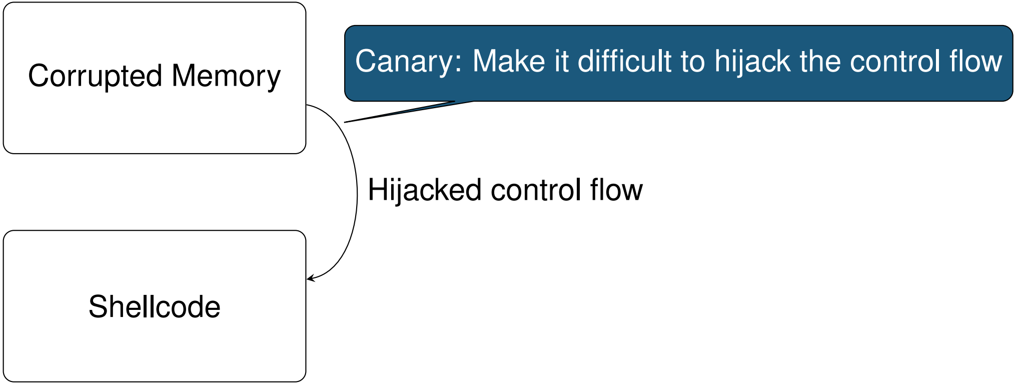
ASLR: Make it difficult to guess the address

Hijacked control flow

Shellcode

DEP: Make this region ***non-executable***

Another Perspective: Canary



Corrupted Memory

Canary: Make it difficult to hijack the control flow

Hijacked control flow

Shellcode

What is canary?

Canary is a bird 🐦.

Canary in a Cole Mine

The bird would act as an early warning for carbon monoxide (CO) gas.

[1]

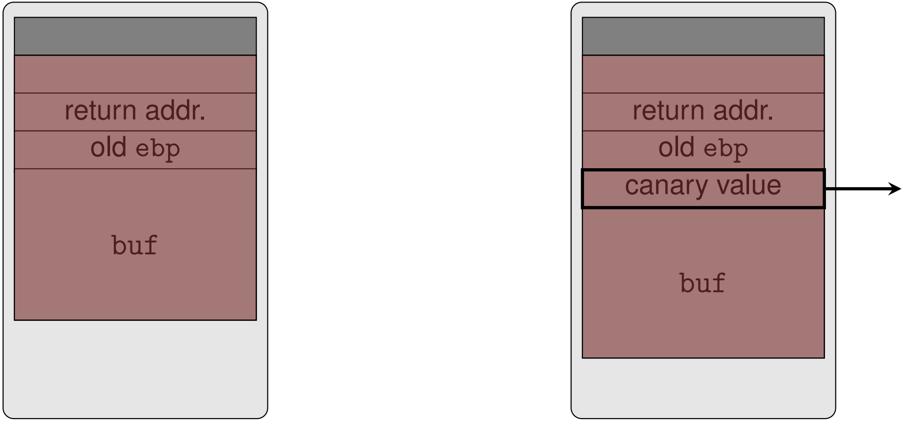
[1] Image from `http://www.academia.dk/Blog/a-canary-in-a-coal-mine-in-the-19th-century-and/`

Canary in Software

- ***Early warnings*** of buffer overflows.
- First introduced in 1998[2].
- Not necessarily used for stack, but can also be used for heap.

[2] StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, ***USENIX Security 1998***.

Stack Canary (a.k.a. Stack Cookie)

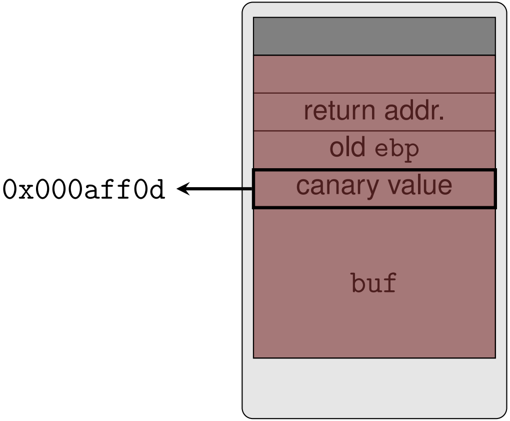


Check before executing return!

w/o Stack Canary

w/ Stack Canary

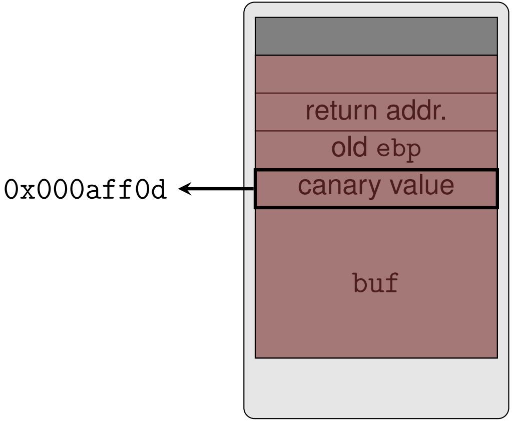
StackGuard (1998)



w/ Stack Canary

StackGuard (1998)

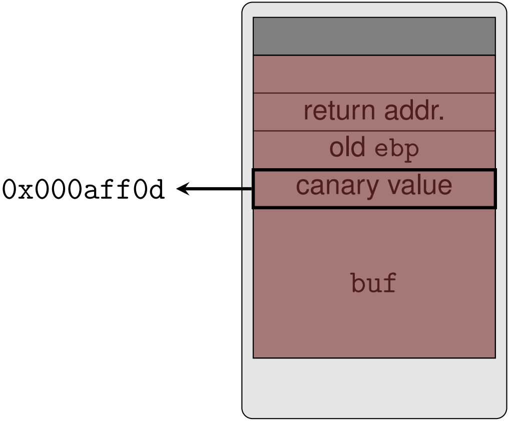
– 0x00 stops strcpy



w/ Stack Canary

StackGuard (1998)

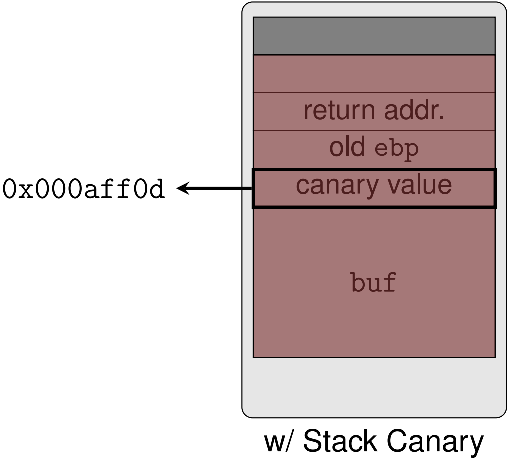
- `0x00` stops `strcpy`
- `0x0a` and `0x0d` stop `fgets`



w/ Stack Canary

StackGuard (1998)

- — 0x00 stops strcpy
- — 0x0a and 0x0d stop fgets
- — 0xff stops EOF checks



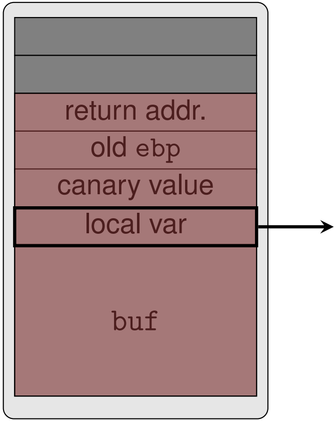
Problem of Using a Constant Canary Value

`memcpy?`

Random Canaries

Pick a random value at process initialization, put it on the stack.

Problem Still Exists



Local variables are not protected!

Reordering Local Variables

- Always put local buffers after local pointers.
- This idea is implemented by GCC 4.1 in 2005.

GCC Stack Canary Implementation

w/o Stack Canary

```
80483fb:  push   ebp
80483fc:  mov    ebp,esp
80483fe:  sub    esp,0x100
8048404:  push   DWORD PTR [ebp+0x8]
8048407:  lea    eax,[ebp-0x100]
804840d:  push   eax
804840e:  call   80482d0 <strcpy@plt>
8048413:  add    esp,0x8
8048416:  leave
8048417:  ret
```

w/ Stack Canary

```
804844b:  push   ebp
804844c:  mov    ebp,esp
804844e:  sub    esp,0x108
8048454:  mov    eax,DWORD PTR [ebp+0x8]
8048457:  mov    DWORD PTR [ebp-0x108],eax
804845d:  mov    eax,gs:0x14
8048463:  mov    DWORD PTR [ebp-0x4],eax
8048466:  xor    eax,eax
8048468:  push   DWORD PTR [ebp-0x108]
804846e:  lea    eax,[ebp-0x104]
8048474:  push   eax
8048475:  call   8048320 <strcpy@plt>
804847a:  add    esp,0x8
804847d:  mov    eax,DWORD PTR [ebp-0x4]
8048480:  xor    eax,DWORD PTR gs:0x14
8048487:  je     804848e <somefn+0x43>
8048489:  call   8048310 <__stack_chk_fail@plt>
804848e:  leave
804848f:  ret
```

GCC Stack Canary Implementation

w/o Stack Canary

```
80483fb:  push   ebp
80483fc:  mov    ebp,esp
80483fe:  sub    esp,0x100
8048404:  push   DWORD PTR [ebp+0x8]
8048407:  lea
804840d:  push
804840e:  call
8048413:  add    esp,0x8
8048416:  leave
8048417:  ret
```

w/ Stack Canary

```
804844b:  push   ebp
804844c:  mov    ebp,esp
804844e:  sub    esp,0x108
8048454:  mov    eax,DWORD PTR [ebp+0x8]
                 DWORD PTR [ebp-0x108],eax
                 eax,gs:0x14
                 DWORD PTR [ebp-0x4],eax
8048466:  xor    eax,eax
8048468:  push   DWORD PTR [ebp-0x108]
804846e:  lea    eax,[ebp-0x104]
8048474:  push   eax
8048475:  call   8048320 <strcpy@plt>
804847a:  add    esp,0x8
804847d:  mov    eax,DWORD PTR [ebp-0x4]
8048480:  xor    eax,DWORD PTR gs:0x14
8048487:  je     804848e <somefn+0x43>
8048489:  call   8048310 <__stack_chk_fail@plt>
804848e:  leave
804848f:  ret
```

Random canary value stored at `gs:0x14`

GCC Stack Canary Implementation

w/o Stack Canary

```
80483fb:  push    ebp
80483fc:  mov     ebp,esp
80483fe:  sub     esp,0x100
8048404:  push    DWORD PTR [ebp+0x8]
8048407:  lea     eax,[ebp−0x100]
804840d:  push    eax
804840e:  call    80482d0 <strcpy@plt>
8048413:  add     esp,0x8
8048416:  leave
8048417:  ret
```

w/ Stack Canary

```
804844b:  push    ebp
804844c:  mov     ebp,esp
804844e:  sub     esp,0x108
8048454:  mov     eax,DWORD PTR [ebp+0x8]
8048457:  mov     DWORD PTR [ebp−0x108],eax
804845d:  mov     eax,gs:0x14
    463:  mov     DWORD PTR [ebp−0x4],eax
     66:  xor     eax,eax
    468:  push    DWORD PTR [ebp−0x108]
804846e:  lea     eax,[ebp−0x104]
8048474:  push    eax
8048475:  call    8048320 <strcpy@plt>
804847a:  add     esp,0x8
804847d:  mov     eax,DWORD PTR [ebp−0x4]
8048480:  xor     eax,DWORD PTR gs:0x14
8048487:  je      804848e <somefn+0x43>
8048489:  call    8048310 <__stack_chk_fail@plt>
804848e:  leave
804848f:  ret
```

Why?

What is GS/FS[3] Segment Register?

- CPU maintains a Local Descriptor Table (LDT) in memory.
- Segment registers hold an offset of the LDT.
- On Linux, GS/FS segment register points to an entry of LDT, which represents a Thread Control Block (TCB).

[3] GS is used on x86, FS is used on x86-64.

TCB and References

TCB structure.

```
typedef struct {
  void *tcb;                /* gs:0x00 Pointer to the TCB. */
  dtv_t *dtv;               /* gs:0x04 */
  void *self;               /* gs:0x08 Pointer to the thread descriptor.  */
  int multiple_threads;     /* gs:0x0c */
  uintptr_t sysinfo;        /* gs:0x10 Syscall interface */
  uintptr_t stack_guard;    /* gs:0x14 Random value used for stack protection */
  uintptr_t pointer_guard;/* gs:0x18 Random value used for pointer protection */
  int gscope_flag;          /* gs:0x1c */
  int private_futex;        /* gs:0x20 */
  void *__private_tm[4];    /* gs:0x24 Reservation of some values for the TM ABI.*/
  void *__private_ss;       /* gs:0x34 GCC split stack support.  */
} tcbhead_t;
```

Who Initializes `gs:0x14`?

Runtime Dynamic Linker (RTLD) initializes it every time it launches a process.

Pseudocode of what RTLD does when initializing a process.

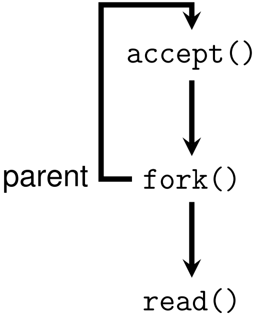
```
uintptr_t ret;
int fd = open("/dev/urandom", O_RDONLY);
if (fd >= 0) {
    ssize_t len = read(fd, &ret, sizeof(ret));
    if (len == (ssize_t) sizeof(ret)) {
        // inlined assembly for moving ret to [gs:0x14]
    }
}
```

GCC Canary (ProPolice) Implementation

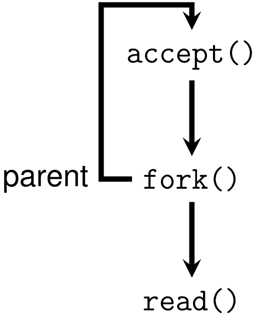
- Use a random canary value for every process creation.
- Puts buffers after any local pointers on the stack.

Attacking Canary Protection

Reused Canary Value



vs.

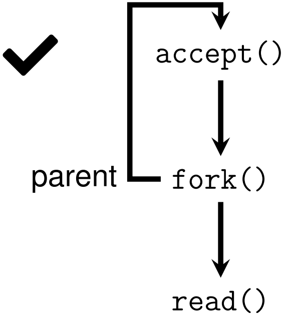


Canary is the same for every child

Canary changes for every child

Reused Canary Value

✓



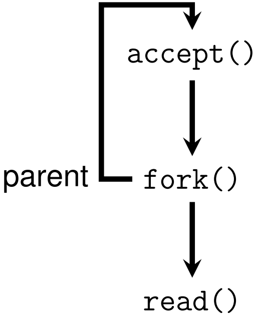
parent

accept()

fork()

read()

vs.



parent

accept()

fork()

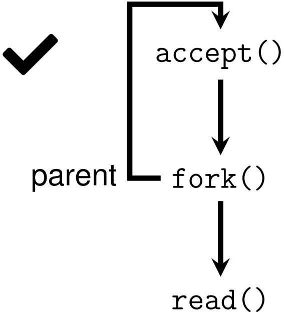
read()

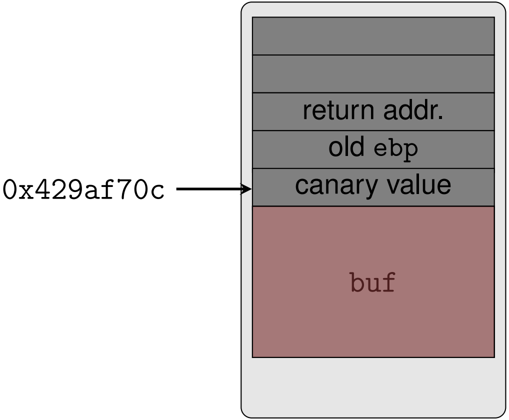
Canary is the same for every child

Canary changes for every child

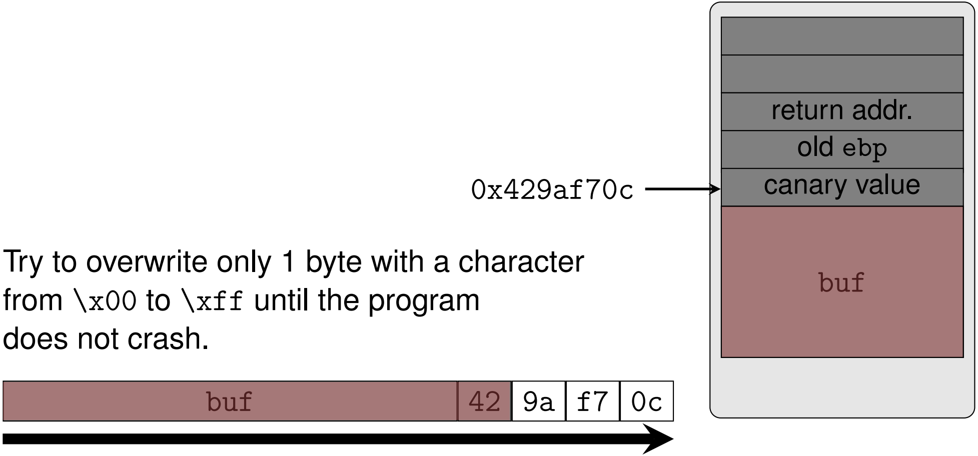
e.g., OpenSSH does this

Attack #1: Byte-by-Byte Brute Forcing





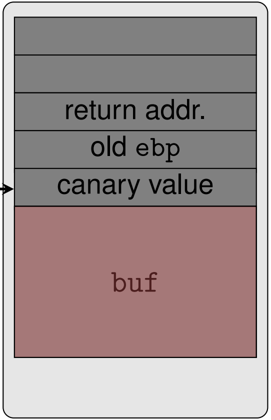
Attack #1: Byte-by-Byte Brute Forcing



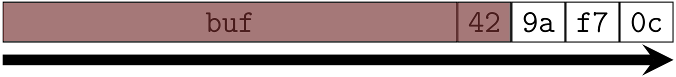
Try to overwrite only 1 byte with a character from `\x00` to `\xff` until the program does not crash.

Attack #1: Byte-by-Byte Brute Forcing

Do the same for all bytes.
Worst case: 256×4 iterations.



Try to overwrite only 1 byte with a character from `\x00` to `\xff` until the program does not crash.



Problems?

Brute-forcing may not work if

1. the canary contains a character that we cannot use, e.g., a NULL byte in canary for `strcpy` overflows.
2. we cannot control the last byte of the buffer.

Example: Uncontrollable Last Byte

```
char *bp = buf;
while (buflen) {
  toread = pr_netio_read(in_nstrm, pbuf->buf,
                          (buflen < pbuf->buflen ? buflen : pbuf->buflen), 1);
  while (buflen && toread > 0 && *pbuf->current != '\n' && toread--) {
...
        if ( *bp == TELNET_IAC ) { /* = 0xFF */
...
          buflen--;
          telnet_mode = 0;
          break;
        }
...
    bp += 1;
    buflen--;
  }
  *bp = '\0';
  return buf;
}
```

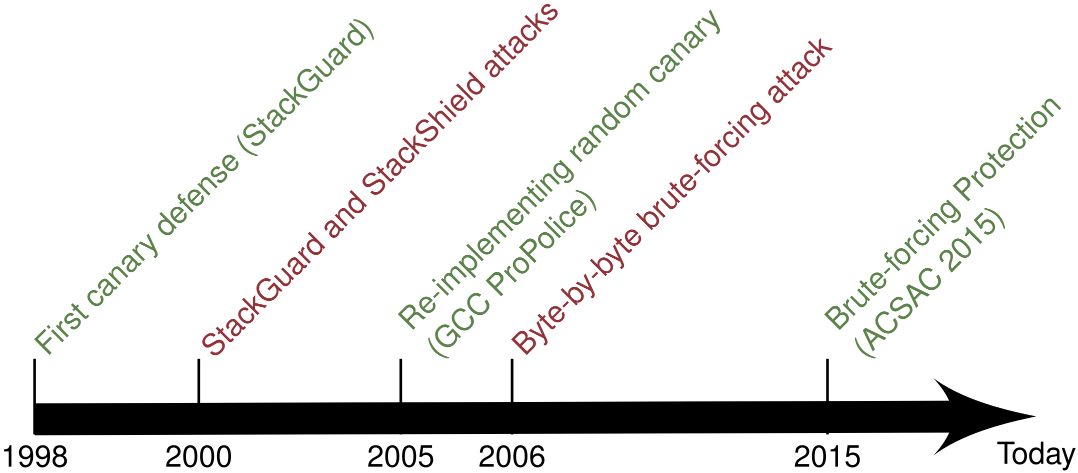
→ Problem: we cannot control the last byte!

ProFTPd (CVE-2010-3867)

Protecting Canary Brute-Forcing Attack

DynaGuard: Armoring Canary-based Protections against Brute-force Attacks, ***ACSAC 2015***.

Canary Attack and Defense Timeline



1998 First canary defense (StackGuard)

2000 StackGuard and StackShield attacks

2005 Re-implementing random canary (GCC ProPolice)

2006 Byte-by-byte brute-forcing attack

2015 Brute-forcing Protection (ACSAC 2015)

Today

Attack #2: Leaking Canary Value

- If there is another vulnerability that allows us to ***leak*** stack contents, then we can easily bypass the canary check.
- Canary is inherently vulnerable to format string attacks.
- Combining memory disclosure with buffer overflow is the next topic.

Question?

Exercise: Revealing Canary Value under GDB

- Create a simple buffer overflow example in C.
- Compile the program with the `fstack-protector` option.
- Read the canary value used for protecting the `main` function.
- See if the canary value varies by re-executing the program under GDB.