OS Security: Challenges, Methods, and, Open Problems

Aravind Machiry

Why security is important for OS?

- Runs in God Mode.

- Control of OS implies:
  - Control of the entire system.
  - Control of all processes.
  - Control of all hardware:
    - **This is changing now!!**

What can go wrong?

- Improper checking/mis-configurations.

Improper checking/mis-configurations

- E.g., Kernel pages having PTE_U bit set.
- Allowing anyone to add themselves as a sudo user.
- Using default passwords!
  - Have you changed your router password?

Improper checking/mis-configurations

Cisco Default Passwords (Valid April 2022)

If you don't see your Cisco device or the default data below doesn't work, see below the table for more help, including what to do.

Cisco Model	Default Username	Default Password	Default IP Address
DPC2320	[none]	[none]	192.168.0.1
ESW-520-24-K9	cisco	cisco	192.168.10.2
ESW-520-24P-K9	cisco	cisco	192.168.10.2
ESW-520-48-K9	cisco	cisco	192.168.10.2

What can go wrong?

- Improper checking/mis-configurations.
- Bad Design Decisions.

Bad Design Decisions

- E.g., Using kernel address as an ID.
  - Can leak memory layout of kernel.

- CVE-2021-27363. Kernel address leak due to pointer used as unique ID.

```
static ssize_t
show_transport_handle(struct device *dev, struct device_attribute *attr,
                      char *buf)
{
        struct iscsi_internal *priv = dev_to_iscsi_internal(dev);
        return sprintf(buf, "%llu\n", (unsigned long long)iscsi_handle(priv->iscsi_transport));
}
static DEVICE_ATTR(handle, S_IRUGO, show_transport_handle, NULL);
```

What can go wrong?

- Improper checking/mis-configurations.
- Bad Design Decisions.
- Bugs in Hardware.

Bugs in Hardware

```
machiry@machirylaptop:~$ cat /proc/cpuinfo | grep bugs
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs taa itlb_multihit srbds
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs taa itlb_multihit srbds
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs taa itlb_multihit srbds
```

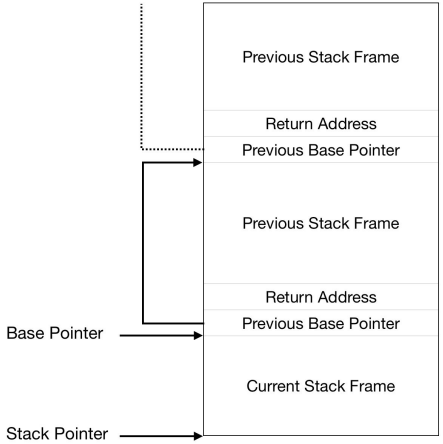
Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. This might include your passwords stored in a password manager or browser, your personal photos, emails, instant messages and even business-critical documents.

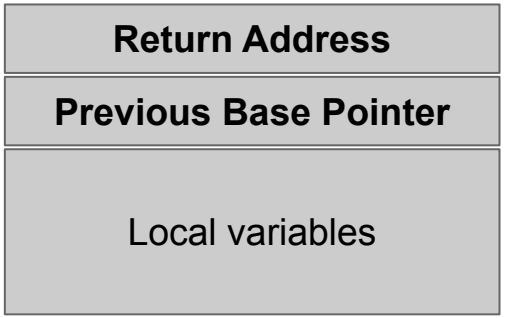
What can go wrong?

- Improper checking/mis-configurations.
- Bad Design Decisions.
- Bugs in Hardware.
- Bugs in the source code - software security.

Can a software bug used to get complete control of the OS?

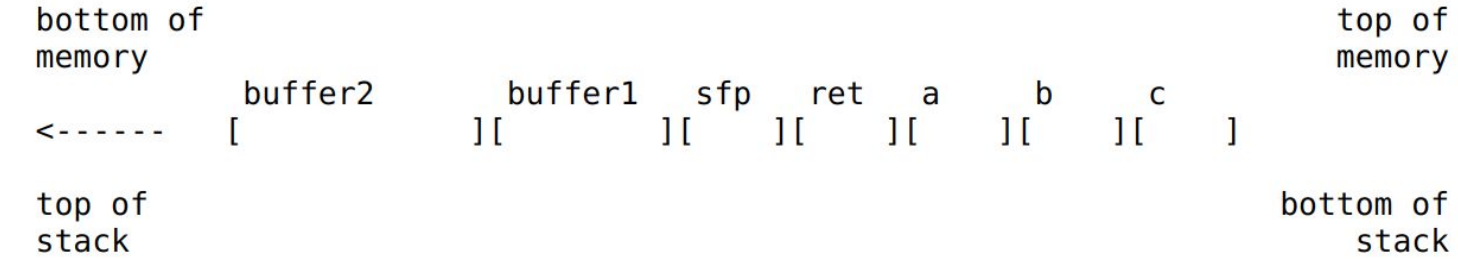
Stack
Primer





Stack Buffer Overflow

```
example1.c:
-------------------------------------------------------------------------------
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
-------------------------------------------------------------------------------
```



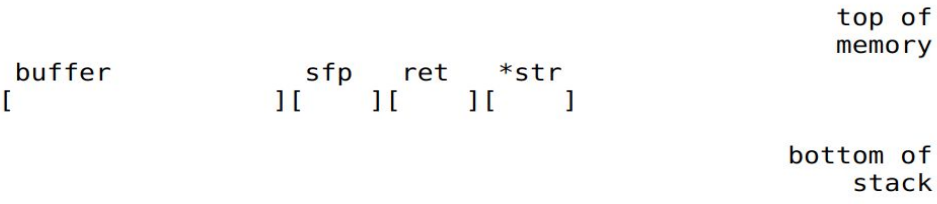
Stack Buffer Overflow

example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

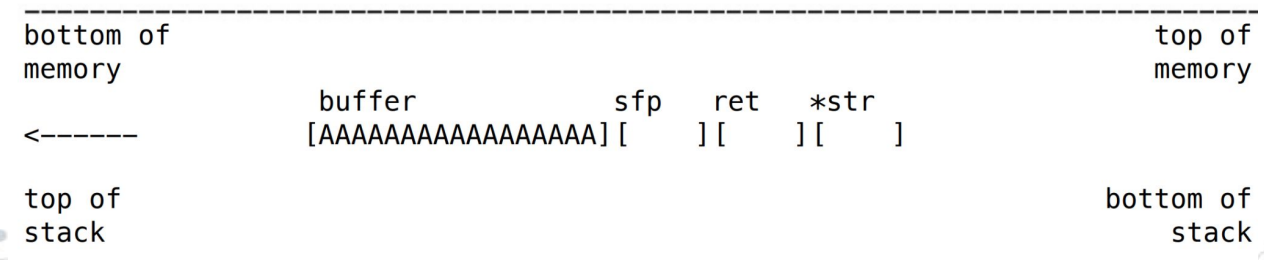
Stack Buffer Overflow

```
example2.c
-------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
-------------------------------------------------------------------------------
```



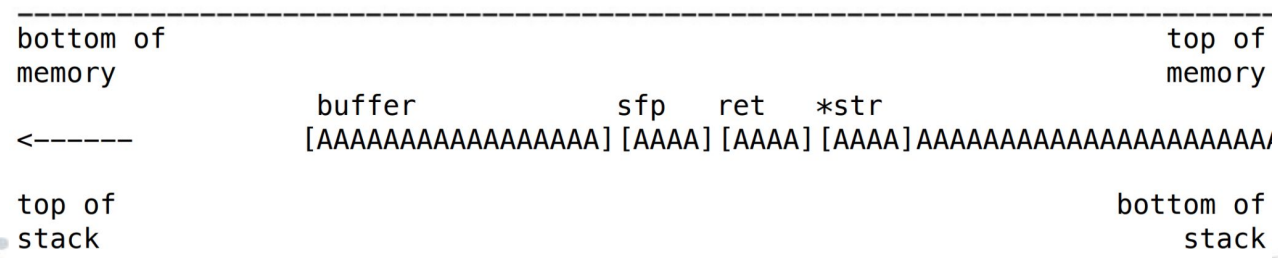
Stack Buffer Overflow

```
example2.c
-------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
-------------------------------------------------------------------------------
```



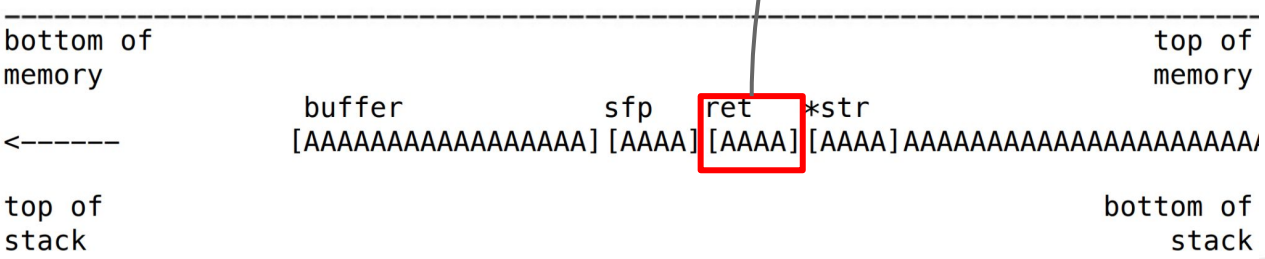
Stack Buffer Overflow

```
example2.c
-------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
-------------------------------------------------------------------------------
```



Stack Buffer Overflow

```
example2.c
-------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
-------------------------------------------------------------------------------
```

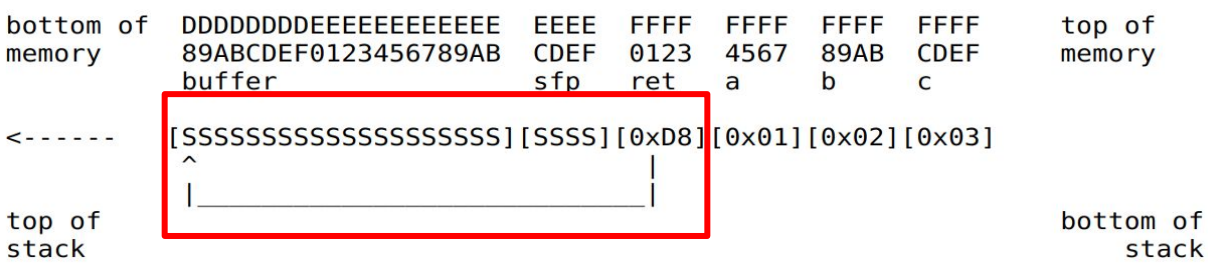


We can control where function returns, i.e., control the execution of the program.

Stack Buffer Overflow

```
example2.c
-------------------------------------------------------------------------------
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
-------------------------------------------------------------------------------
```

We can make return address point to the data we just provided (SSSS..), i.e., make the program execute our code.



Integer Overflow

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;       /* [1] */

    if(size > len){             /* [2] */
        return -1;
    }

    memcpy(out, buf1, size1);
    memcpy(out + size1, buf2, size2);

    return size;
}
```

Integer Overflow

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    return size;
}
```

Read 2 pieces of data from socket.

Integer Overflow

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }
    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));
    return size;
}
```

Convert first 4 bytes into integers.

Integer Overflow

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;       /* [1] */

    if(size > len){             /* [2] */
        return -1;
    }

    return size;
}
```

Add both the numbers and check that sum is less than len.

Integer Overflow

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;       /* [1] */

    if(size > len){             /* [2] */
        return -1;
    }

    memcpy(out, buf1, size1);
    memcpy(out + size1, buf2, size2);

    return size;
}
```

Copy the data into out buffer.

What's wrong?

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;       /* [1] */

    if(size > len){             /* [2] */
        return -1;
    }

    return size;
}
```

Say len = 16

What happens when:
size1 = 0x7fffffff
size2 = 0x7fffffff

What's wrong? Integer Overflow

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;       /* [1] */

    if(size > len){             /* [2] */
        return -1;
    }

    return size;
}
```

Say len = 16

What happens when:
size1 = 0x7fffffff
size2 = 0x7fffffff
size = (0x7fffffff + 0x7fffffff = 0xfffffffe (-2)) and (-2 < 16)

Integer Overflow

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;       /* [1] */

    if(size > len){             /* [2] */
        return -1;
    }

    memcpy(out, buf1, size1);
    memcpy(out + size1, buf2, size2);

    return size;
}
```

Say len = 16

Overflowing the buffer pointed by out

We are copying 2*0x7fffffff bytes into out whose size is just 16.

Integer Overflow

```
int get_two_vars(int sock, char *out, int len){
    char buf1[512], buf2[512];
    unsigned int size1, size2;
    int size;

    if(recv(sock, buf1, sizeof(buf1), 0) < 0){
        return -1;
    }
    if(recv(sock, buf2, sizeof(buf2), 0) < 0){
        return -1;
    }

    /* packet begins with length information */
    memcpy(&size1, buf1, sizeof(int));
    memcpy(&size2, buf2, sizeof(int));

    size = size1 + size2;       /* [1] */

    if(size > len){             /* [2] */
        return -1;
    }

    memcpy(out, buf1, size1);
    memcpy(out + size1, buf2, size2);

    return size;
}
```

Say len = 16

Overflowing the buffer pointed by out

What if out is on stack? **Stack based buffer-overflow!!**

We are copying 2*0x7fffffff bytes into out whose size is just 16.

Importance of Software Security

- Most often even small software bugs can lead to severe security vulnerabilities (i.e., complete program control or arbitrary code execution).

bpf: fix incorrect sign extension in check_alu_op()

```
Distinguish between
BPF_ALU64|BPF_MOV|BPF_K (load 32-bit immediate, sign-extended to 64-bit)
and BPF_ALU|BPF_MOV|BPF_K (load 32-bit immediate, zero-padded to 64-bit);
only perform sign extension in the first case.

Starting with v4.14, this is exploitable by unprivileged users as long as
the unprivileged_bpf_disabled sysctl isn't set.

Debian assigned CVE-2017-16995 for this issue.
```

```
diff --git a/kernel/bpf/verifier.c b/kernel/bpf/verifier.c
index 625e358ca765e..c086010ae51ed 100644
--- a/kernel/bpf/verifier.c
+++ b/kernel/bpf/verifier.c
@@ -2408,7 +2408,13 @@ static int check_alu_op(struct bpf_verifier_env *env, struct bpf_insn *insn)
                         * remember the value we stored into this reg
                         */
                        regs[insn->dst_reg].type = SCALAR_VALUE;
-                       __mark_reg_known(regs + insn->dst_reg, insn->imm);
+                       if (BPF_CLASS(insn->code) == BPF_ALU64) {
+                               __mark_reg_known(regs + insn->dst_reg,
+                                                insn->imm);
+                       } else {
+                               __mark_reg_known(regs + insn->dst_reg,
+                                                (u32)insn->imm);
+                       }
                }
```

What type of software security issues can occur in OSes?

- Memory corruption => Control of Execution.
  - Spatial:
    - Buffer overruns and underruns.
  - Temporal:
    - Use-after-free, Double free, etc.

- Race Conditions => Memory corruption.
- Denial of Service (DoS).
- Many others.

Security Issues Specific to Kernel

- Double fetch bugs (CVE-2016-6130)

```
68   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                        sizeof(*sccb))) {
69     rc = -EFAULT;
70     goto out_free;
71   }
72   if (sccb->length > PAGE_SIZE || sccb->length < 8)
73     return -EINVAL;
74   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                        sccb->length)) {
75     rc = -EFAULT;
76     goto out_free;
77   }
...
81   if (copy_to_user(u64_to_uptr(ctl_sccb.sccb), sccb,
                      sccb->length))
82     rc = -EFAULT;
...
86 }
```

Security Issues Specific to Kernel

- Double fetch bugs

First copy: we check the user provided length.

```
68   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                        sizeof(*sccb))) {
69     rc = -EFAULT;
70     goto out_free;
71   }
72   if (sccb->length > PAGE_SIZE || sccb->length < 8)
73     return -EINVAL;
74   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                        sccb->length)) {
75     rc = -EFAULT;
76     goto out_free;
77   }
...
81   if (copy_to_user(u64_to_uptr(ctl_sccb.sccb), sccb,
                      sccb->length))
82     rc = -EFAULT;
...
86 }
```

Security Issues Specific to Kernel

- Double fetch bugs

Second copy: we are overriding the checked length.

```
68    if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                         sizeof(*sccb))) {
69      rc = -EFAULT;
70      goto out_free;
71    }
72    if (sccb->length > PAGE_SIZE || sccb->length < 8)
73      return -EINVAL;
74    if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                         sccb->length)) {
75      rc = -EFAULT;
76      goto out_free;
77    }
...
81    if (copy_to_user(u64_to_uptr(ctl_sccb.sccb), sccb,
                       sccb->length))
82      rc = -EFAULT;
...
86  }
```

Security Issues Specific to Kernel

- Double fetch bugs

We are using the new length without checking.

```
68   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                        sizeof(*sccb))) {
69     rc = -EFAULT;
70     goto out_free;
71   }
72   if (sccb->length > PAGE_SIZE || sccb->length < 8)
73     return -EINVAL;
74   if (copy_from_user(sccb, u64_to_uptr(ctl_sccb.sccb),
                        sccb->length)) {
75     rc = -EFAULT;
76     goto out_free;
77   }
...
81   if (copy_to_user(u64_to_uptr(ctl_sccb.sccb), sccb,
                      sccb->length))
82     rc = -EFAULT;
...
86 }
```

Security Issues Specific to Kernel

- ## Reference Counting Bugs:

  Reference counting (refcount) has become a default mechanism that manages resource objects.

  A refcount of a tracked object is incremented when a new reference is assigned and decremented when a reference becomes invalid.

  Due to the inherent complexity of the kernel and resource sharing, developers often fail to properly update refcounts, leading to refcount bugs.

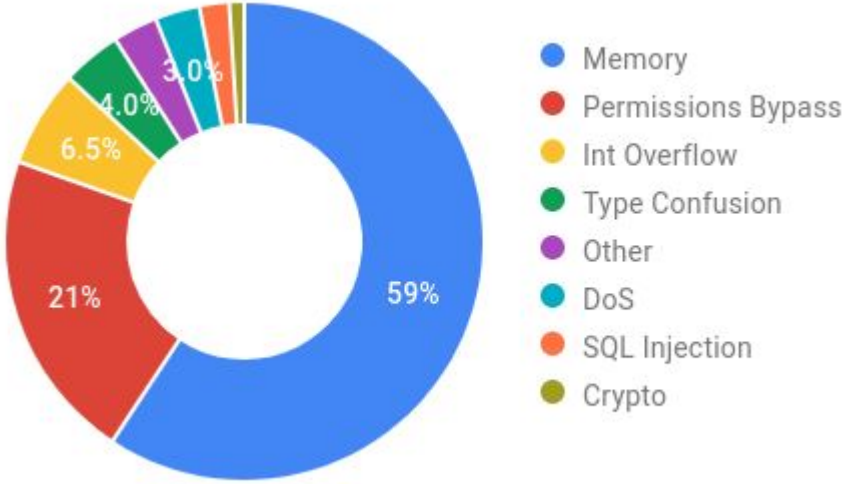
  Researchers have shown that refcount bugs can cause critical security impacts like privilege escalation

Security Issues Specific to Kernel

- Reference Counting Bugs:

```
1 struct inode *ext4_orphan_get(struct super_block *sb, ...)
2 {
3     ...
4     struct buffer_head *bitmap_bh = NULL;
5     // increase refcount if success
6     bitmap_bh = ext4_read_inode_bitmap(sb, block_group);
7     if (IS_ERR(bitmap_bh))
8         return ERR_CAST(bitmap_bh);
9     ...
10    inode = ext4_iget(sb, ino, EXT4_IGET_NORMAL);
11    if (IS_ERR(inode)) {
12        // missing refcount decrease here
13        ...
14        return inode;
15    }
16    ...
17    brelse(bitmap_bh);  // decrease refcount
18    return inode;
19 }
```

Distribution



How can we fix this?

- Use memory safe languages: Java, Rust, Go, etc
  - Legacy code!?

- Find and Fix bugs.

How can we fix this?

- Use memory safe languages: Java, Rust, Go, etc
  - Legacy code!?

- Find and Fix bugs.

Finding Bugs

- Static Analysis
- Dynamic Analysis

Static Analysis

- Analyze the code

Use grep!

```
struct kernel_obj ko;

void update_value(int *ptr) {
     *ptr +=1;
}

void entry_point(void *uptr, int len){
     curr_date->item = &ko;

     copy_from_user(&ko, uptr, len);

     for (int i=0; i < ko.count; i++) {
         update_value(&(ko.data[i]));
     }
     strcpy(gbuff, curr_data->buf);
     strcat(gbuff, curr_data->item);
     kobject_put(curr_data->item);
}
```

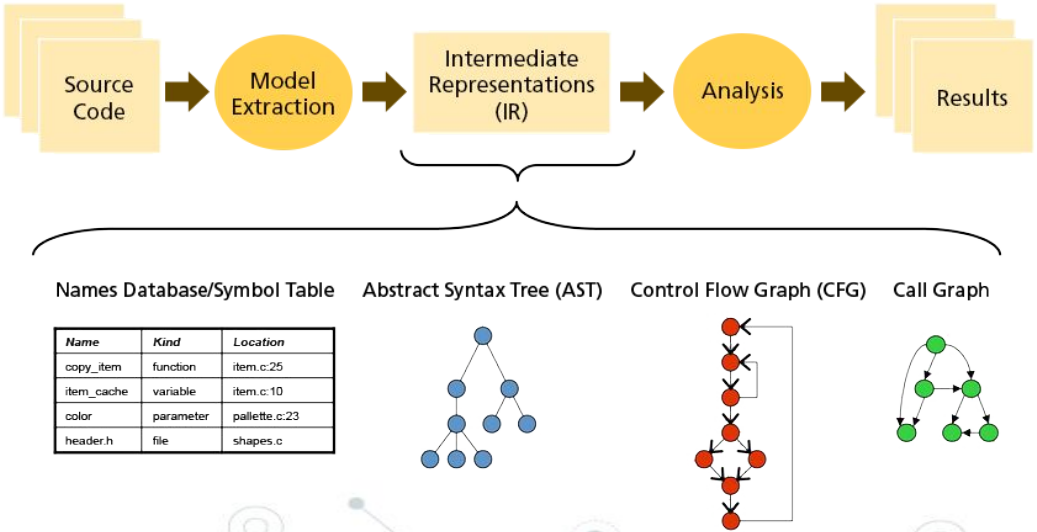
Calls to dangerous function

Lot of false positives!

	CppCheck	flawfinder	RATS	Sparse
Qualcomm	18	4,365	693	5,202
Samsung	22	8,173	2,244	1,726
Huawei	34	18,132	2,301	11,230
MediaTek	168	14,230	3,730	13,771
Total	**242**	**44,990**	**8,968**	**31,929**

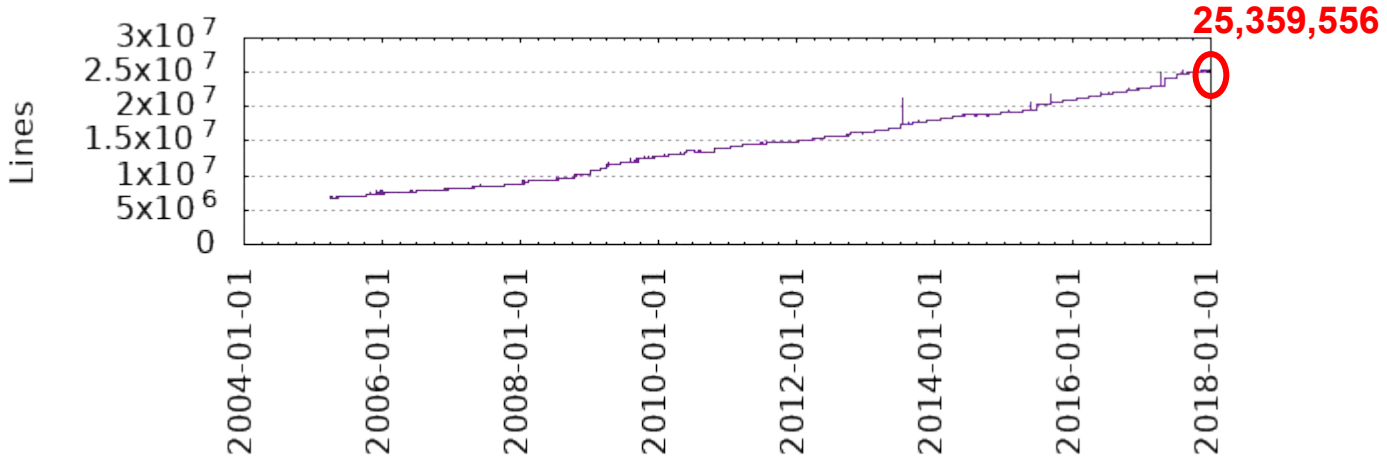
Static Analysis

- We can use sophisticated analysis:
  - Dataflow analysis, Taint tracking, etc.



Hard to Scale

Kernel code is large and complex



Static Analysis: Best effort techniques

- Heuristics and light weight techniques:
  - Coverity: https://scan.coverity.com/
  - lgtm: https://lgtm.com/

- Machine Learning:
  - BRAN (Research paper)

Static Analysis Drawbacks

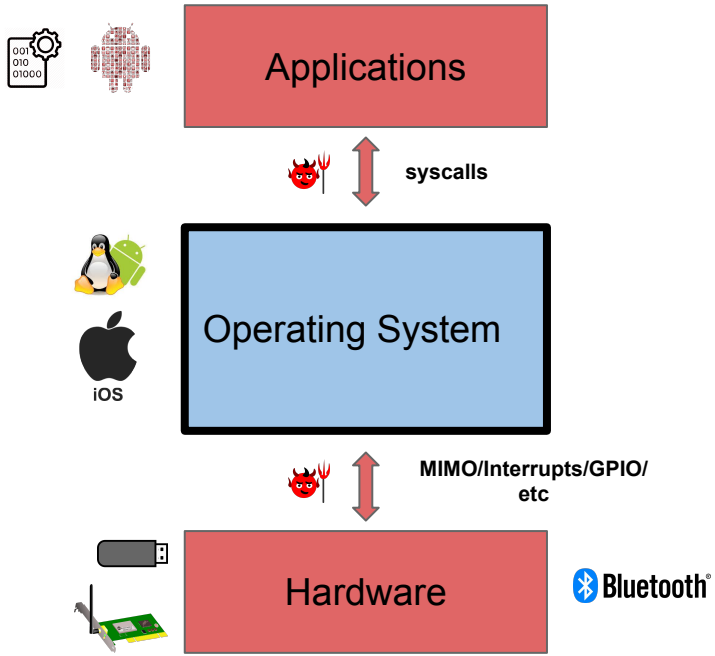
- False positives:
  - The warnings do not always mean real security bugs.

Dynamic Analysis/Testing

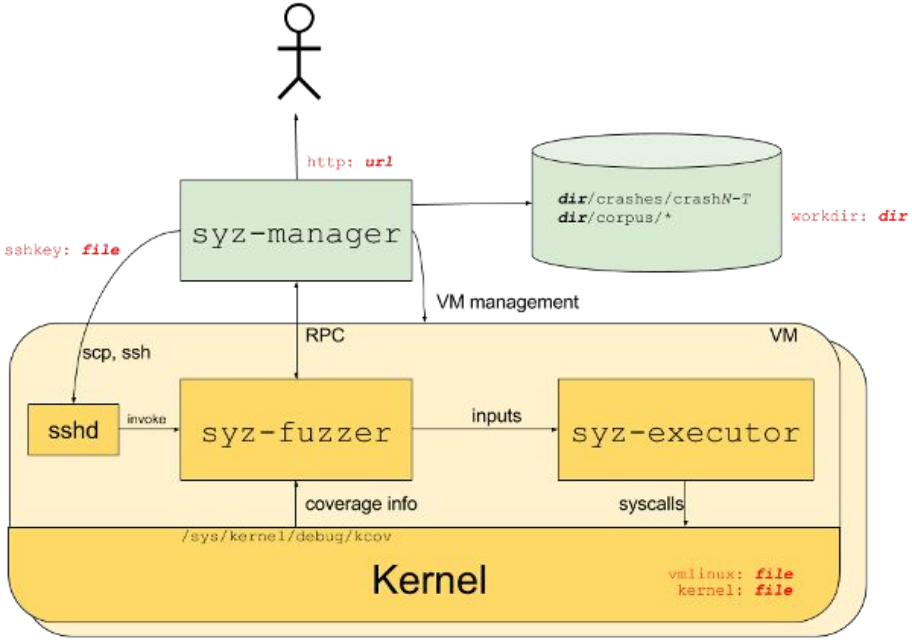
- Automated Testing:
  - Automatically generate test cases and run program with these test cases.
  - *Fuzzing*: provide random data to OS and see if OS crashes.

Fuzzing OS

- How to provide input to OS?
  - How to interact with OS?
    - From user space: System calls
    - From hardware: MIMO/Interrupts



Syzkaller: Fuzzer for System calls



Syzkaller

- Continuous automated testing of Linux kernel:
  - https://syzkaller.appspot.com/upstream

- Many improvements are on-going!

Current State of OSes

Malicious photo app exploits Android kernel vulnerability

John Leyden 08 January 2020 at 16:46 UTC
Updated: 13 May 2020 at 05:27 UTC

Mobile | Malware | Vulnerabilities

Kernel vulnerabilities in Android devices using Qualcomm chips explored

Updated: The security flaws that allowed attackers to achieve root capabilities on handsets have now been described in detail.

Current State of OSes

Critical iOS bug could have given hackers complete control of your iPhone over Wi-Fi

Google's team of security analysts, Project Zero first published a report flagging the flaw termed as unauthenticated kernel memory corruption vulnerability.

Written by **Shriparna Saha**
December 3, 2020 1:24:50 pm

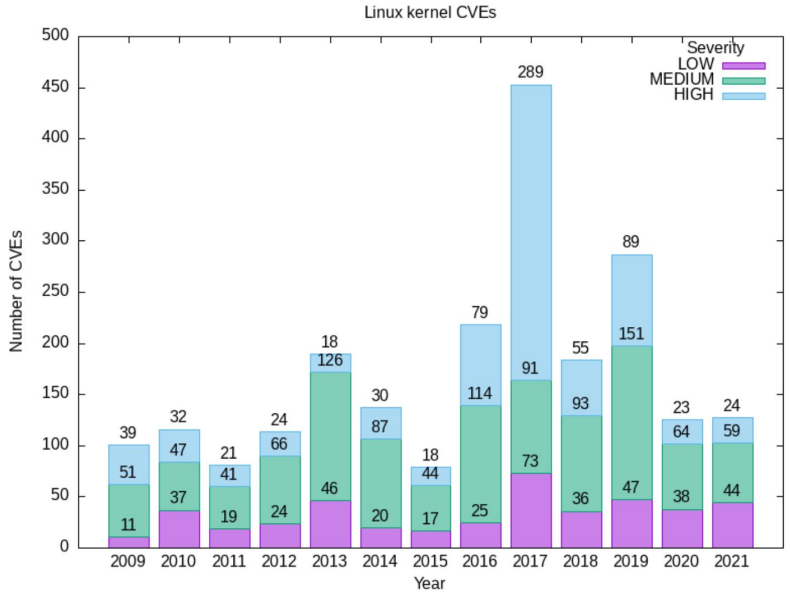
Current State of OSes

Windows kernel zero-day vulnerability used in targeted attacks

By **Sergiu Gatlan**

October 30, 2020 01:38 PM 0

Current State of Linux Kernel



Protecting your OS

- Follow good software engineering practices:
  - Design Reviews
  - Code Reviews
  - Dedicated Test Team/Red Team

Protecting your OS

- Follow good software engineering practices.

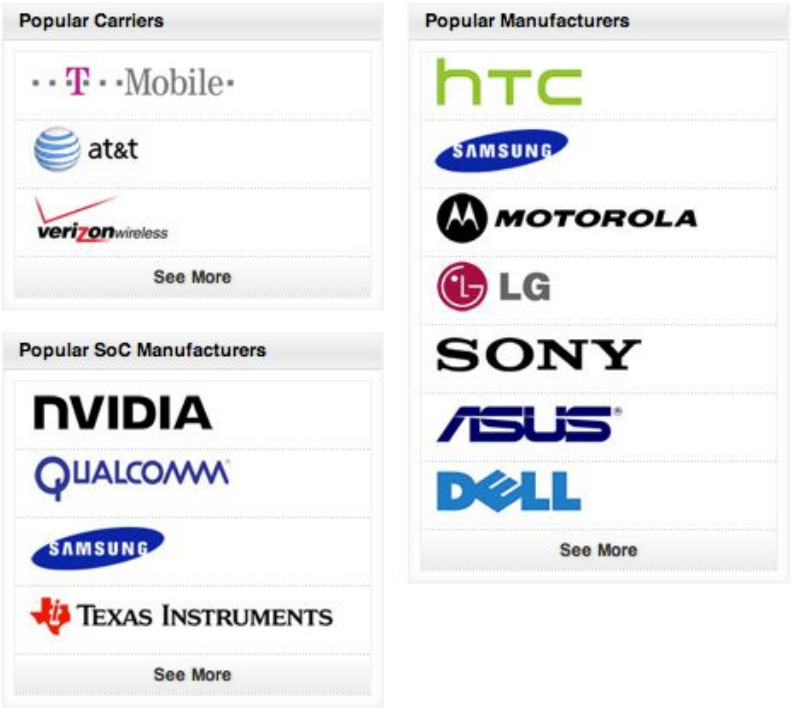
- Use automated static analysis tools:
  - lgtm: https://lgtm.com/
  - Coverity: https://scan.coverity.com/

Protecting your OS

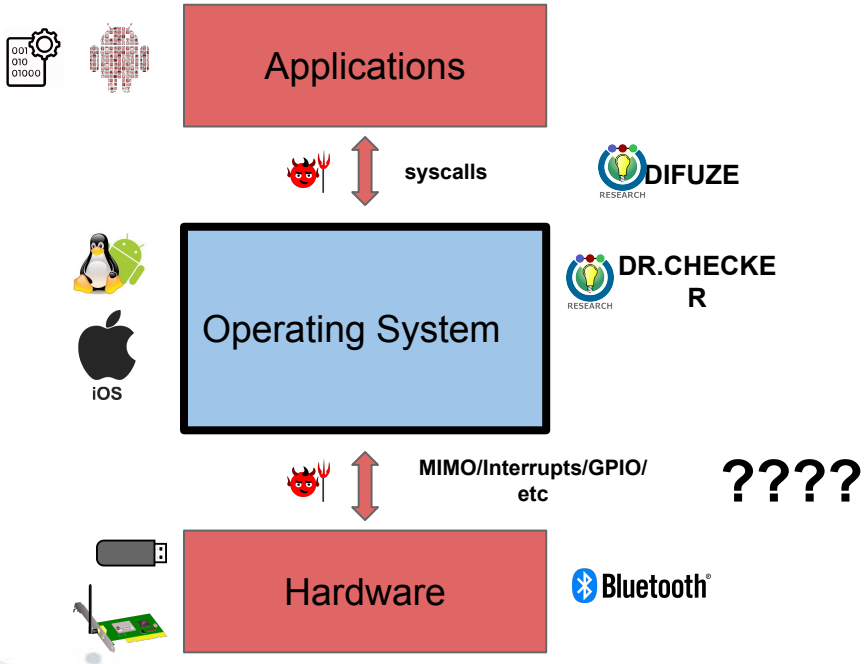
- Follow good software engineering practices.
- Use automated static analysis tools.
- Try using automated fuzzing tools:
  - Syzkaller
  - Peach fuzzer
  - etc.

Fuzzing Challenge: Device Independent Fuzzing

- ~100 different versions of Linux kernels used in different commercial devices.
- Fuzzing require access to devices. **Can we fuzz without real devices?**



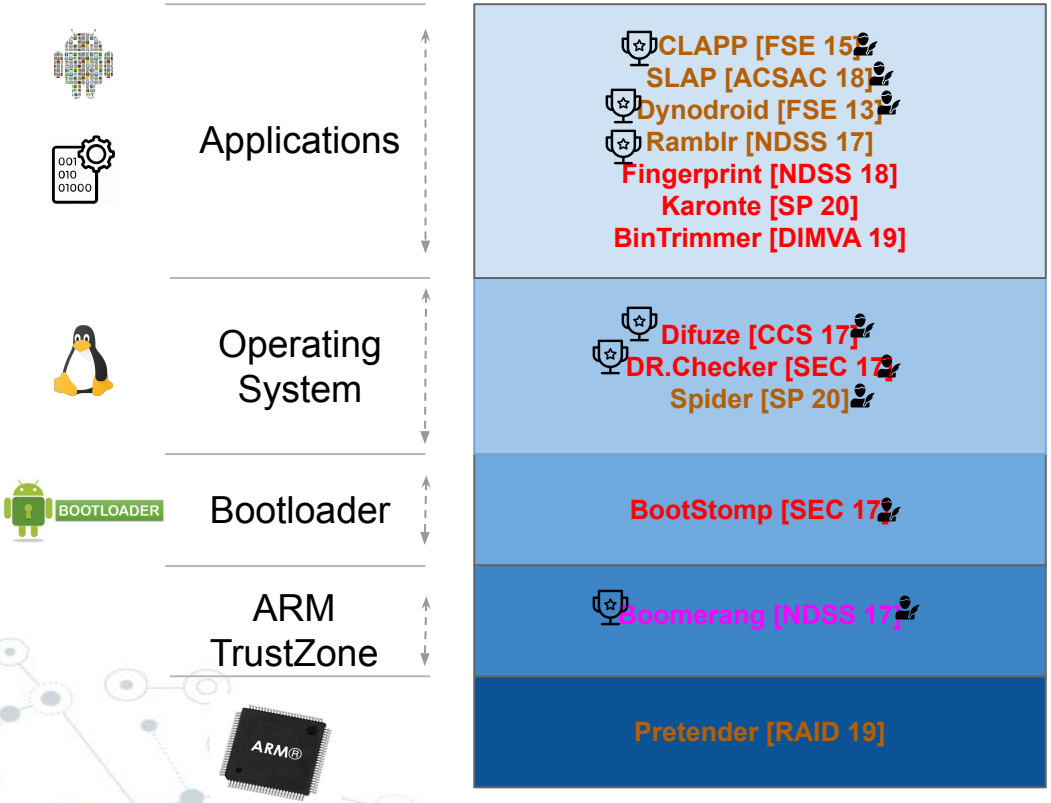
Can we fuzz from Hardware?



PurS3 Lab

Top tier:

- SP (IEEE, Oakland)
- SEC (USENIX)
- NDSS (ISOC)
- CCS (ACM)



Awards and Recognitions

Idea / first author

- New Capabilities
- Vulnerability Detection
- New attacks and defenses

Want to Secure OSes?

- **Actively looking for Undergrads/M.S/PhD students.**
- http://purs3lab.github.io/
- amachiry@purdue.edu


The End