February 22, 2016

Advanced Exploits

Computer Security 2 (GA02)

Emiliano De Cristofaro, Gianluca Stringhini

Thanks to Giovanni Vigna and Christopher Kruegel for some of the material

Windows Exploitation

Windows runs on ~90% of the world's computers

How do attackers exploit Windows machines?

We saw that on Linux shellcodes rely on the invocation of system calls (0x80 instruction in assembly)

On Windows, the system call interface (0x2e or sysenter) is poorly documented and changes between versions → **Not reliable**

Attackers use library functions instead (**Windows API**)

All Windows programs link against the **kernel32.dll library**

kernel32.dll contains two important functions that allow us to execute any function

- LoadLibraryA(libraryname)
- GetProcAddress(hmodule, functionname)

Problem: **An attacker has to find their address first!**

Locating kernel32

Windows allocates a Process Environment Block (PEB) structure for every running process – **its address is at a fixed offset in the process memory**

The PEB structure contains three linked lists with information about the loaded modules that have been mapped in the process space

- One of the lists is ordered by the initialization time
- kernel32.dll is always the second module to be initialized

It is possible to extract the base address for kernel32.dll from the PEB! – kernel32.dll can then **load additional libraries**

Locating GetProcAddress

DLLs include an **image export directory** section (.edata) from which it is possible to get the address of GetProcAddress.

Calling this function, it is possible to retrieve the address of the function to call in the shellcode

Popular choices are

- Download / Execute
  - kernel32.dll: CreateFile, CreateProcessA
  - wininet.dll: InternetOpenUrlA, InternetReadFile
- Reverse shell (remote shell)
  - Kernel32.dll: CreateProcessA
  - ws2_32.dll: WSASocketA, connect, bind, listen, accept

Windows Buffer Overflow Prevention

Stack-Based OverRun Detection – basically stack canaries

Data Execution Prevention – we already saw this

Address Space Layout Randomization – we already saw this

SAFESEH – We saw that an attacker could overwrite the pointer to the Structured Exception Handler (SEH) to point to his own code. SAFESEH aims at preventing this

Whenever an exception is raised, the exception is transferred to a handler in ntdll.dll, which checks if the exception target belongs to a list of whitelisted addresses (stored in the PE header of the program).

Linux Buffer Overflow Prevention

Similar to the Windows counterpart:

- **Address Space Layout Randomization**
- **Executable Stack Protection**
- **Stack Smashing Protection**

Additional countermeasures:

- **Fortify Source** – compiler countermeasure, provides runtime checks of buffer lengths and memory regions
- **Propolice** – stronger stack canary protection that places byte arrays right next to the canary

February 29, 2016

Advanced Exploits

Computer Security 2 (GA02)

Emiliano De Cristofaro, Gianluca Stringhini

Thanks to Giovanni Vigna and Christopher Kruegel for some of the material

Some recap

In memory, data, control information and code are mixed

An attacker can craft his input to **overwrite important control information** such as the return address of a function and **hijack the program execution**

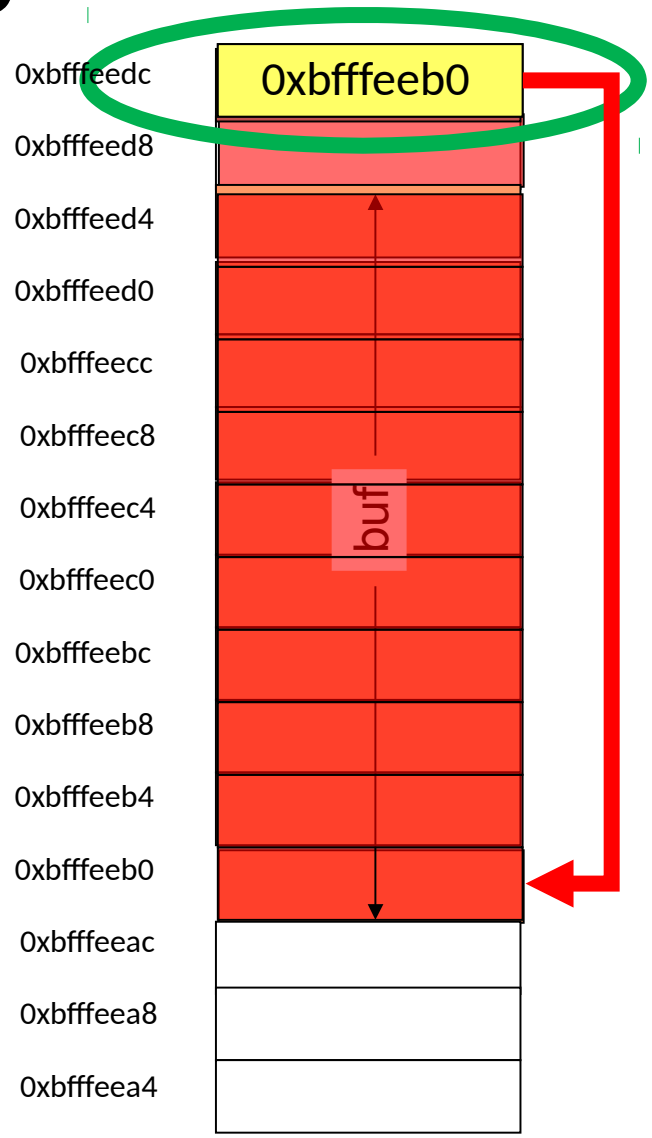
Buffer overflows come in many fashions

- Stack overflow – jump to shellcode
- Return-into-libc
- Return Oriented Programming (ROP)

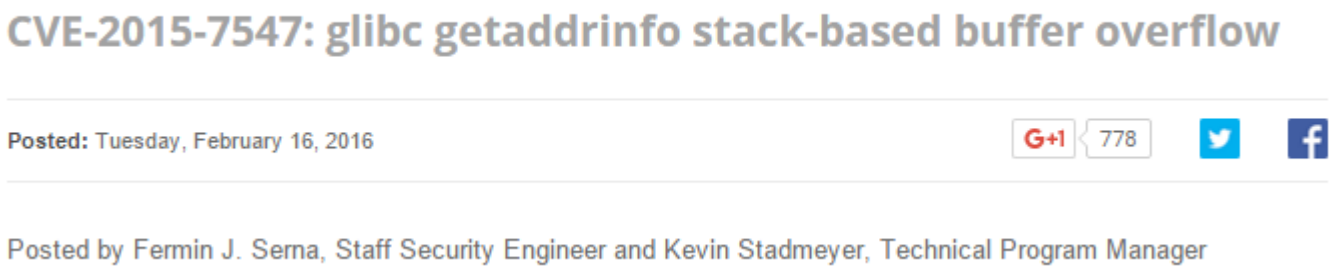
Multiple countermeasures have been developed

- Stack canaries
- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Control Flow Integrity

Still, the problem of buffer overflows has not been solved completely



Recent news: the glibc DNS stack overflow vulnerability



Attackers could set up a DNS under their control and issue DNS replies that exploit the vulnerability

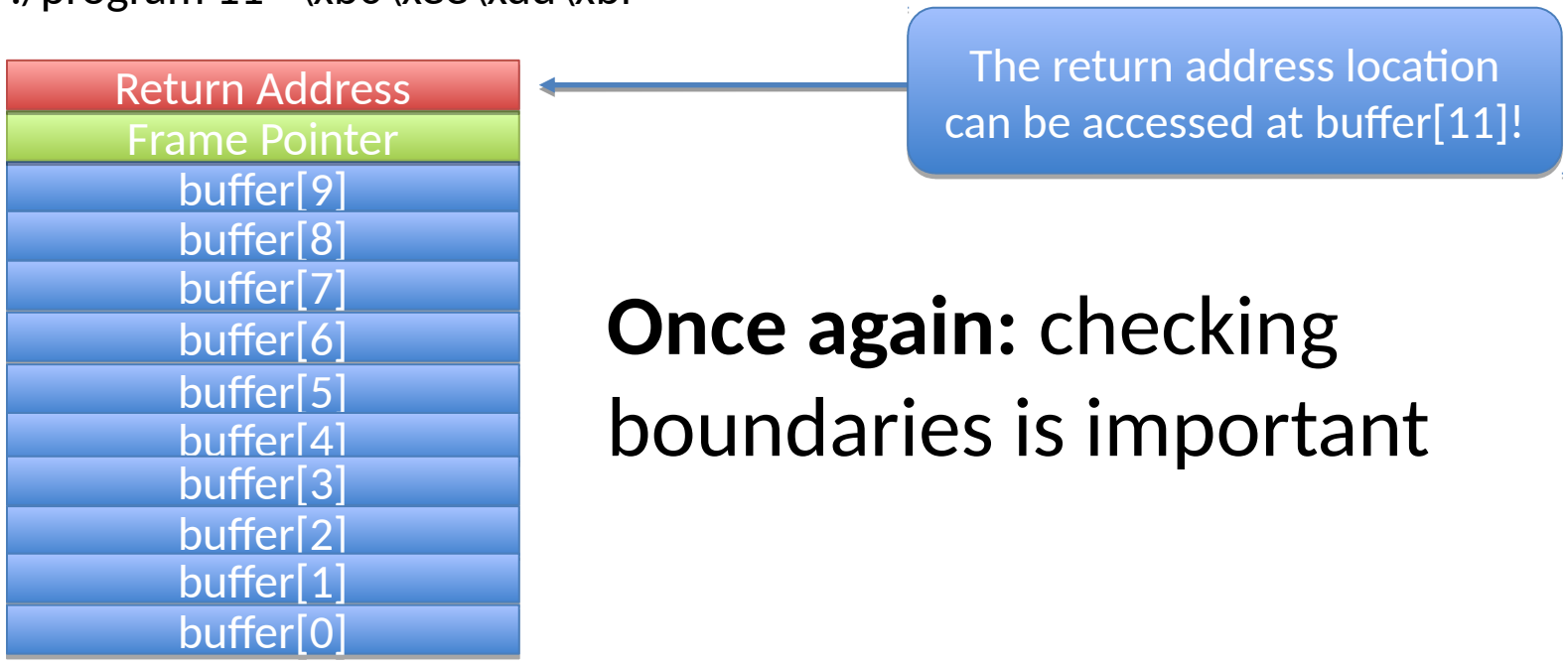
Array Overflows

Consider the following code snippet:

```
int buffer[10];
buffer[atoi(argv[1])] = argv[2];
```

What happens if the user calls the program as follows:

./program 11 “\xb0\xee\xdd\xbf”



Once again: checking boundaries is important

Non-terminated String Overflows

Sometimes **buffers are too small** to store a shellcode.

Some functions, such as strncpy(), limit the amount of data copied in the destination buffer but do not include a terminator (i.e., ‘\0’) once the limit is reached.

If adjacent buffers are not null terminated too an attacker could chain them together, storing part of the shellcode in each of them.

Integer Overflows

Integer variables can only reach a certain value (2,147,483,647 for a signed int32 variable).

What happens if we try to increment its value further?

We end up with a **very large negative number**.

Other problems can come from errors in casting values:

```
unsigned long l; short x = -2; l = x;
```

(l is now 4,294,967,294)

Integer overflows can have unexpected security implications

Several implementations of the SSH1 protocol were affected by an integer overflow vulnerability.

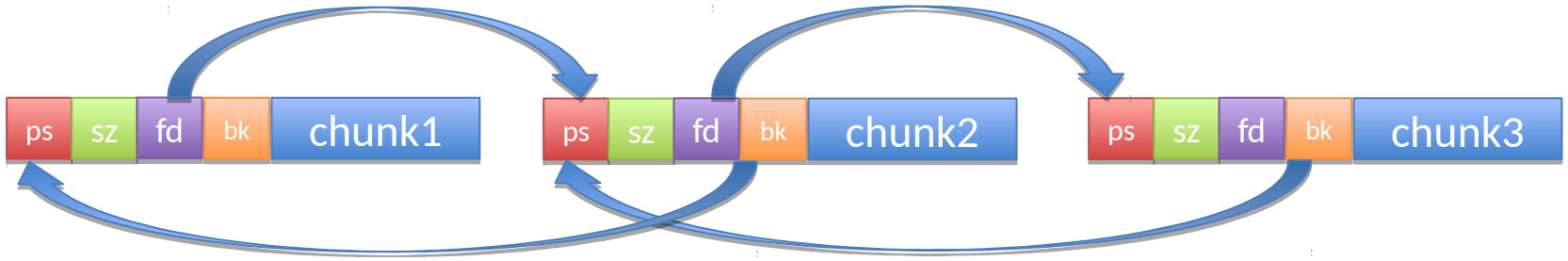
- A function used a dynamically allocated hash table whose size was determined by user input
- Attackers could make the **input overflow and generate a hash table of size 0**
- When the program attempted to store values in the hash table of size 0, these values could be used to overwrite the function return address

Heap Overflows

The **heap** is the area of memory that is dynamically allocated through the **`malloc()`** family of functions.

Unlike the stack, the heap grows towards higher memory addresses.

Each memory chunk is stored in the heap and contains both management information (size of previous chunk, size of current chunk, next chunk, previous chunk) and user data.



Simple Heap Overflow

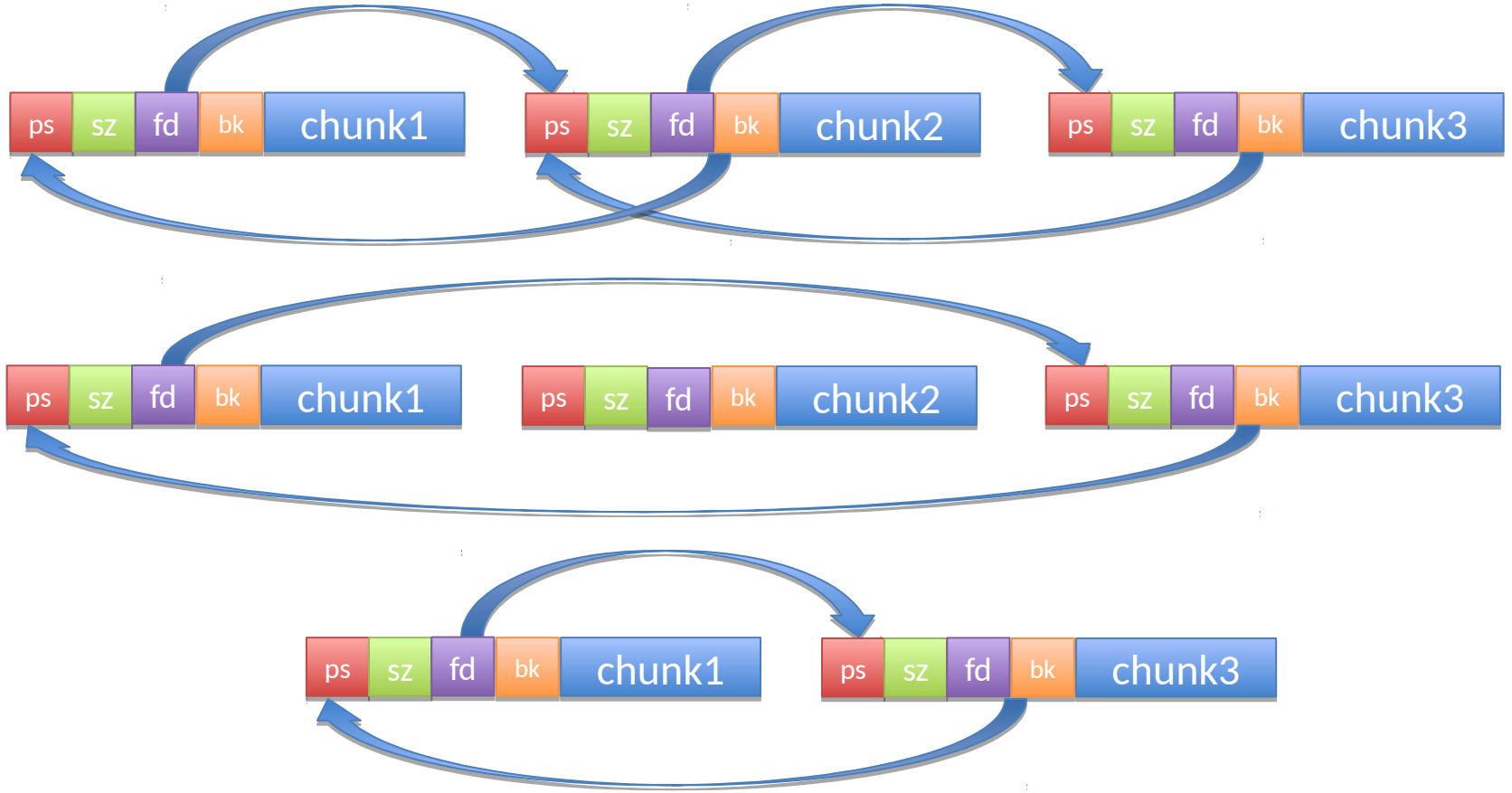
```
struct data {
  char name[64];
};
struct fp {
  int (*fp)();
};
int main(int argc, char **argv) {
  struct data *d;
  struct fp *f;
  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = exit;
  strcpy(d->name, argv[1]);
  f->fp();
}
```

Shellcode is executed!

The data and fp structures are stored in adjacent memory chunks, and the strcpy function can be used to overwrite the function pointer in fp.

Unlinking a Chunk

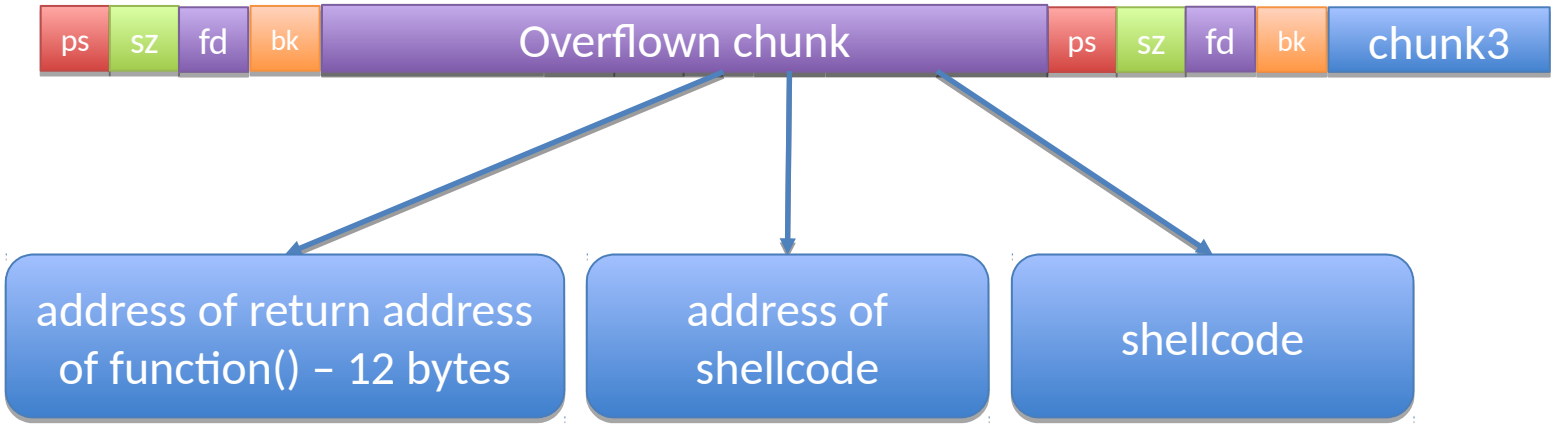
When chunk2 is marked as free (by calling a `free()` function), the chunk is removed from the heap



Overwriting Heap Pointers

When unlinking chunk2, the following operation happens:

```
(chunk2->fd)->bk = chunk2->bk
```



Once function() returns, the execution will be transferred to the shellcode.

Double Free Vulnerabilities

We saw that memory chunks on the heap can be deallocated by calling the **`free()`** function.

If a programmer makes a mistake and **frees the same memory location twice**, this typically breaks the program.

Sometimes, however, a double free can allow an attacker to hijack the program execution.

What the attacker can do is storing a fake memory chunk in the freed location. When **`free()`** is called the second time, the same operations described for the chunk unlinking will be performed.

Use-after-free vulnerabilities

When a chunk of memory gets freed, it potentially becomes available to be allocated again.

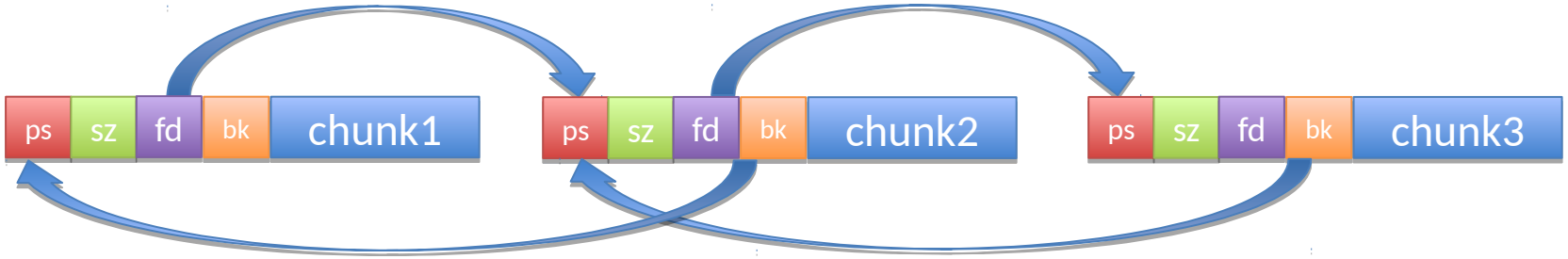
An attacker could get a hold of that chunk, and put malicious data in there.

Once the program accesses the memory region again, it will access the malicious data instead of what the programmer put there.

Preventing Heap Overflows

Safe unlinking: at the time of freeing a chunk, the following check is performed:

Chunk → Next → Previous == Chunk → Previous → Next == Chunk



This condition should always hold. If it does not, it means that an attacker tampered with the addresses in the memory chunk.

Heap entry header cookie: an 8-bit random value is added to the header of each heap chunk and checked when the entry is freed. Similar to **stack canaries**.

Heap spraying

The heap stores dynamically allocated variables and, unlike the stack, **survives the lifetime of functions**.

It looks therefore a good place to store shellcode.

Attackers use a technique known as heap spraying: they fill the heap with copies of their shellcode, and then jump to it by using another exploit (for example a stack buffer overflow).

Having many copies of a shellcode in memory increases the chances of successful exploitation.

Format String Vulnerabilities

`printf` is a weird function

```
printf("Hello World!\n");
printf("%d",string);
printf(buffer);
```

These are good uses of printf

Buffer could be interpreted as a format string! What if buffer = "%d %d"?

If the arguments for the format string are not provided, values on the stack are used instead
→ **an attacker can read the program's memory**!

When a %n is found in a format string, the number of output characters printed so far is stored at the address passed as argument.

An attacker can use a format string to overwrite any address in memory!

Solution: **escaping all "%" characters** when passing strings to a **`printf`**.

TOCTTOU Attacks

Stands for **Time of Check to Time of Use**.
Programs often check information on the filesystem (hard drive) to make decisions on what to do → Example: **check if a file is writable, then write on it**.

Program execution and filesystem operations are not synchronized!

```
if (access("file", W_OK) != 0) {
  exit(1);
}                                    symlink("/etc/passwd", "file");
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

An attacker can wait for the program to check the permission on the file, change where the file points to before the file is open, and force the program to overwrite a system file (e.g., the password database)!

In 2004, Dean et al. showed that there is no portable, deterministic technique to prevent TOCTTOU attacks.

Possible mitigations include **locking files** or including **file system transactions** to the filesystem.

Type confusion attacks

We discussed how languages such as **Java** are **memory safe** and therefore exempt from memory corruption vulnerabilities.

Other vulnerabilities exist though. The most common is a **type confusion vulnerability**.

In a type confusion attack, a flaw in one of the Java Virtual Machine components makes it possible for an object to have multiple types, one of which could allow privileged operations on data.

Type confusion attacks can break the affected language's security (e.g., allowing the attacker to read private fields).

The kernel is a program too

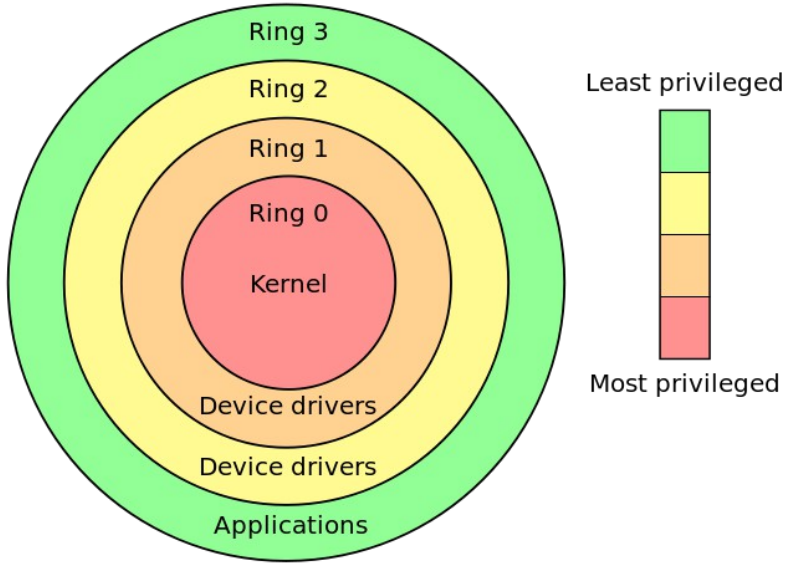
The kernel is the “heart” of the operating system

- Scheduling
- Device drivers

Protection rings

The kernel runs at Ring 0, which has the highest privileges.

An attacker exploiting a kernel vulnerability can accomplish anything, including disabling the security countermeasures that we described before (ASLR, DEP).



Kernel exploitation

Kernel space is more complex than the program memory space.

Some things make the attacker's life easier

- **Programs have their own memory space**, and jumping outside the program's memory space causes a segmentation fault (program crash)
- **Kernel memory does not have a process associated to it**, therefore a kernel exploit can jump anywhere within the memory space

Some things make the attacker's life harder

- Any time the CPU is in ring0, the kernel code is said to be executing in **interrupt context**, meaning that the code can't block (i.e., wait for input)
- Kernel memory is limited (1GB on Linux), and the stack is small (4KB for 32 bit Linux)
- Libraries might not be available

For these reasons, kernel exploits typically **install the payload code in user space**, and then switch to ring3 – This process is usually done through a **stager**.

Stager

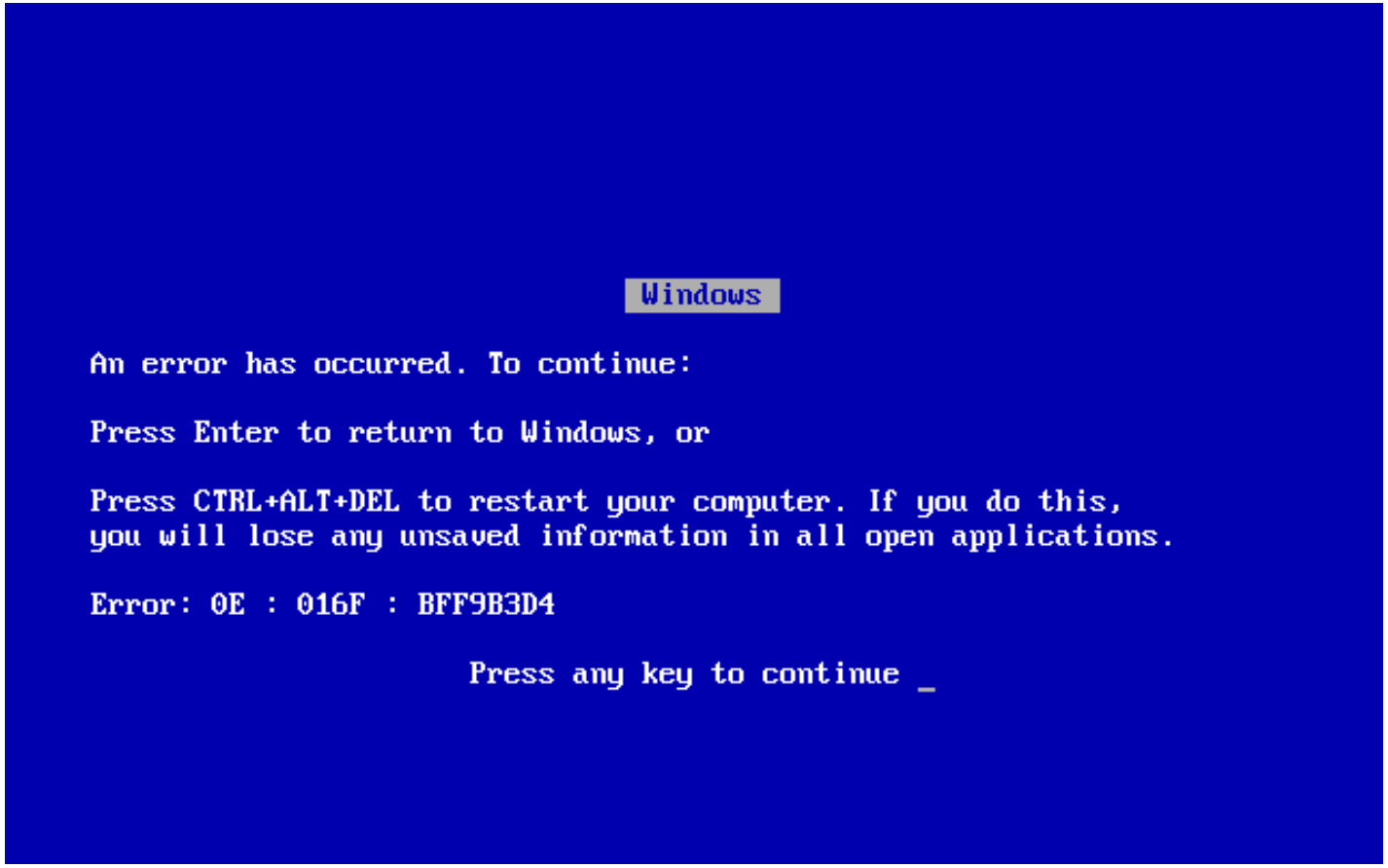
The exploit installs a hook that executes the payload later

- **System call hook** – the system call dispatcher is substituted with a pointer to the payload
- **Thread notify routine** – the routine that is called every time a thread is created is substituted with a pointer to the payload
- **Object type initializer** – the routine that is called every time a certain object type is created is substituted with a pointer to the payload

If the kernel crashes after the stager completed its job, the exploit fails too.

Kernel exploit need to **restore from the exploit** and keep the state in a safe situation.

In kernel space, recovery is important



Kernel vulnerabilities

- NULL dereference
- Heap or SLAB overflows
- Stack overflows
- Logical bugs

NULL dereference

Normally, when a program tries to access a NULL pointer it crashes.

Why? Because it tries to jump to memory address 0 (0x00000000).

The memory address 0 is a **valid memory address**, and lies in the lower part of the user address space → **The kernel can access it**!

When the kernel is invoked by a program calling a system call, the kernel does not switch address space, but “reuses” the one of the program.

NULL dereference exploit:

- Map valid code to address 0
- Trigger null pointer dereference in the kernel
- The kernel will **execute the code** with higher privileges!

Example payload: set the privileges of the current process to root.

HEAP or SLAB overflows

The Linux kernel can dynamically allocate memory by calling the **`kmalloc()`** function.

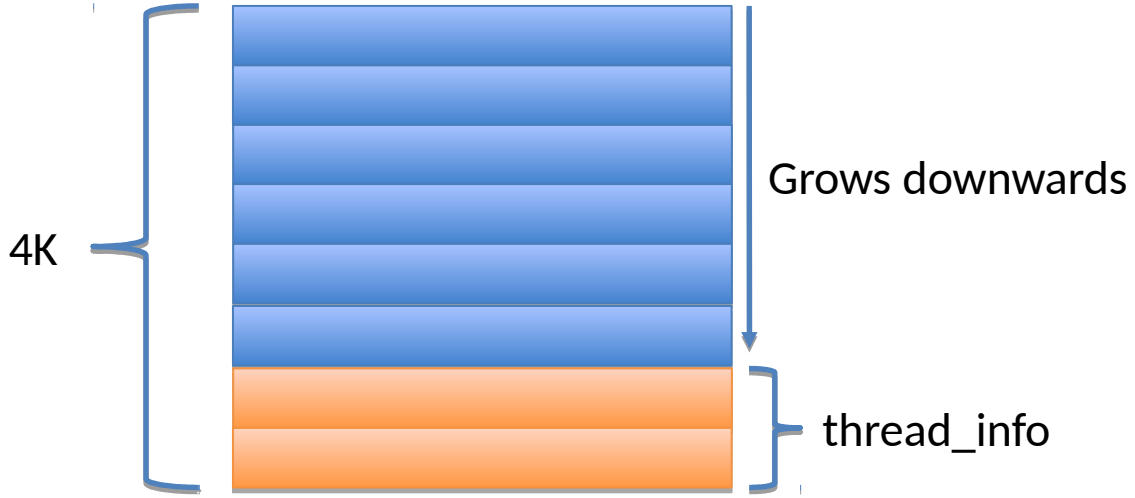
The SLAB allocator of the Linux kernel allows it to dynamically allocate small memory objects – **it works similarly to the heap**.

An attacker can overflow the chunk allocated to her, and **overwrite a structure containing function pointers** in the adjacent SLAB.

More recent Linux versions use a different memory allocator (SLUB), it is more efficient but has the same security issues.

Kernel stack overflow

The kernel has a stack too, **one stack per kernel thread**

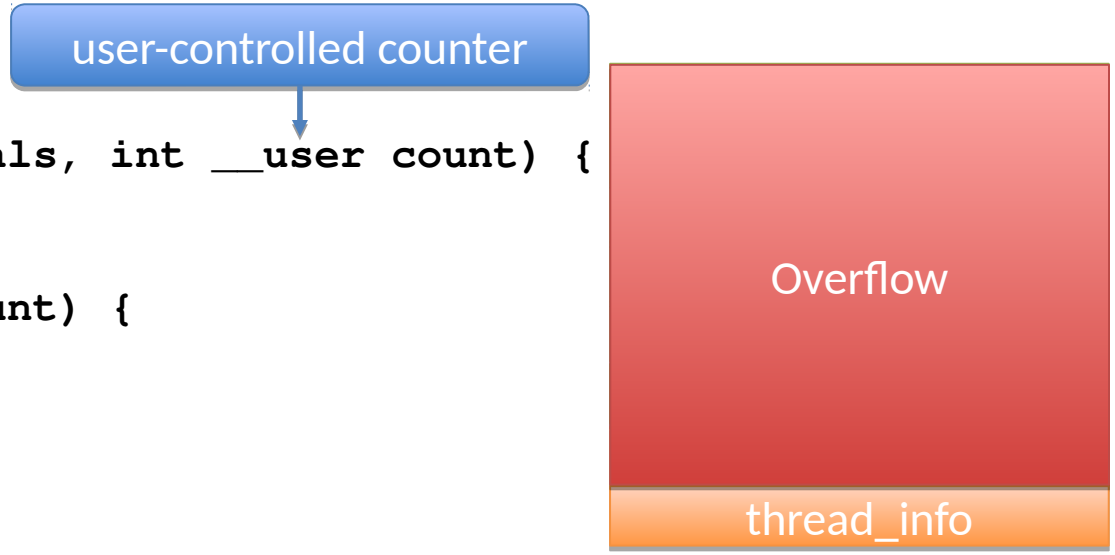


An attacker can overwrite variables, return addresses, or function pointers on the stack

Problem: the stack in the kernel is small (4K on x86, 8K on X86_64) → **no space for shellcode**

Kernel stack overflow – an example

```
static int blah(int __user *vals, int __user count) {
  int i;
  int big_array[count];
  for (i = 0; i < count; ++count) {
     big_array[i] = vals[i];
  }
}
```



An attacker can use this code to overwrite the **`thread_info`** section.

The **`restart_block`** structure contained in **`thread_info`** contains a function pointer that is called any time a **`SYS_restart_syscall`** is invoked from user space.

The attacker can then have the function pointer point to a function of his choice and invoke his exploit by calling.

`syscall(SYS_restart_syscall)` from the user space.

Kernel logical vulnerabilities

Logical vulnerabilities don't have to do with memory corruption directly, but rather with the wrong understanding by programmers of what the program is supposed to do or with the hardware architecture – **race conditions**.

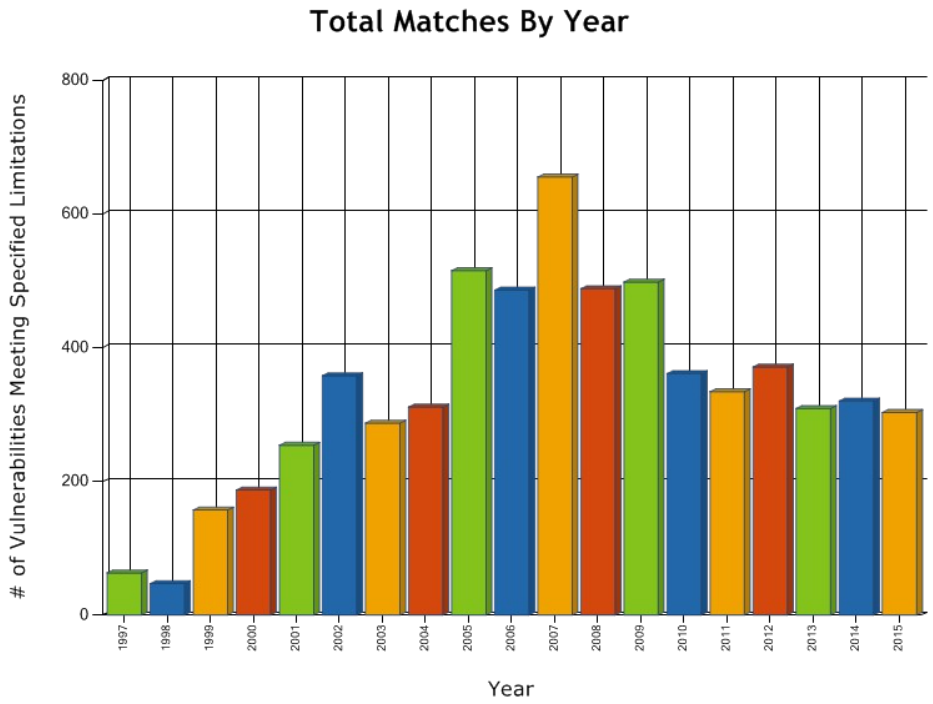
Memory corruption vulnerabilities are "easy" to find, and can be easily fixed, the same is not true for logical vulnerabilities.

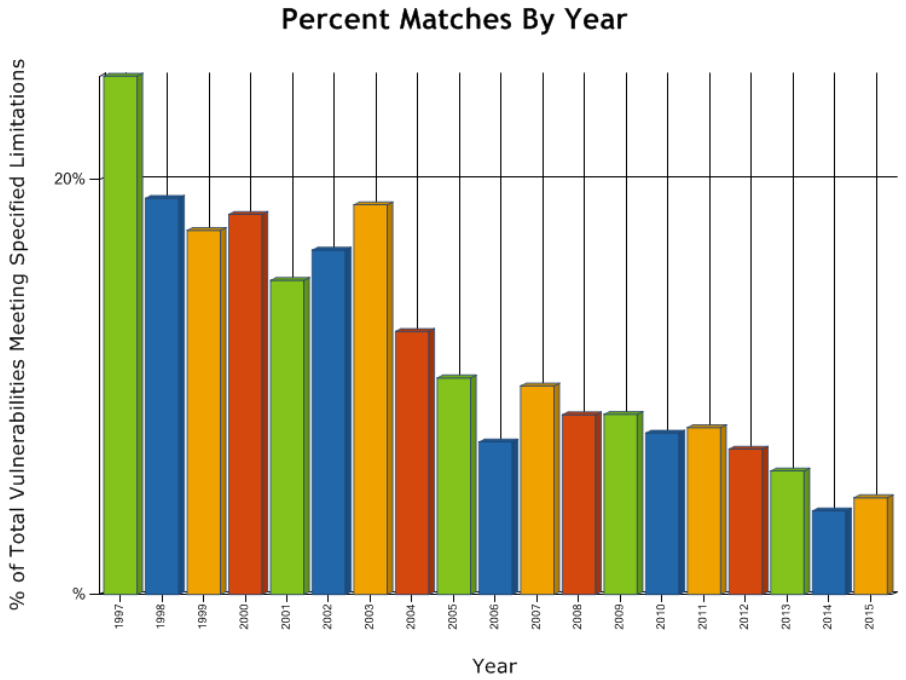
If a memory page is not cached, a "**page fault**" is raised an the active thread is put to sleep before the page becomes available in the cache.

An attacker can modify that memory page before the thread is waken up, and potentially hijack the kernel program flow by doing that.

An attacker can "force" the page fault to happen by filling the memory cache from a different process.

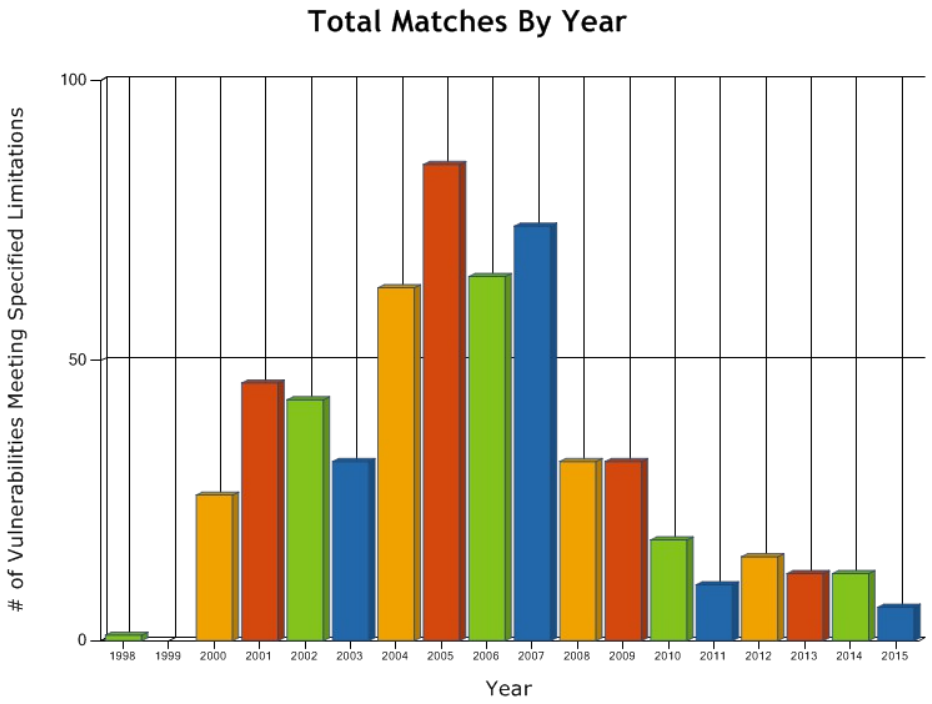
Some stats – Buffer overflows

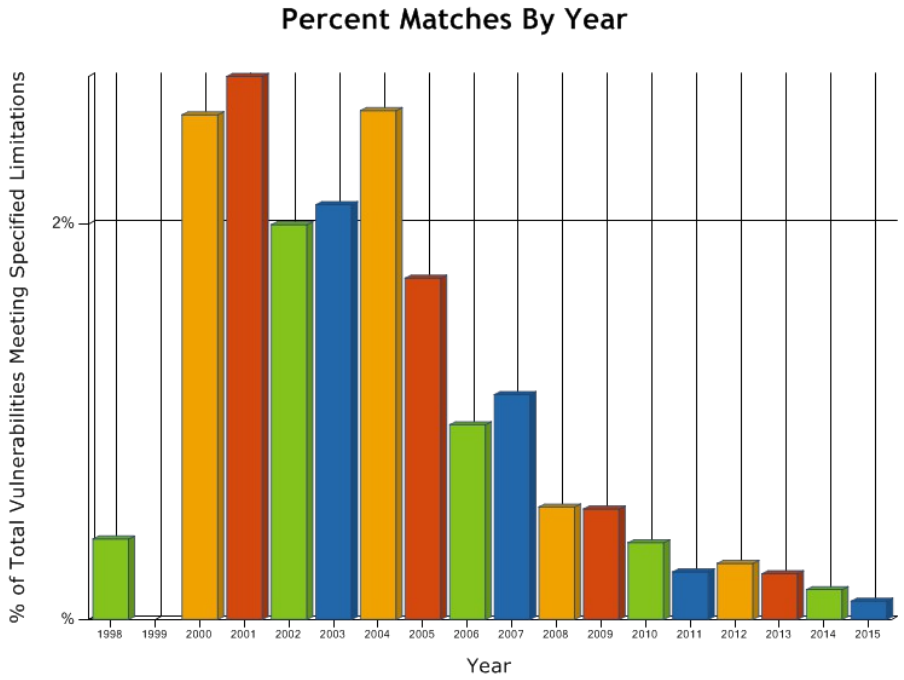




Source: NIST website

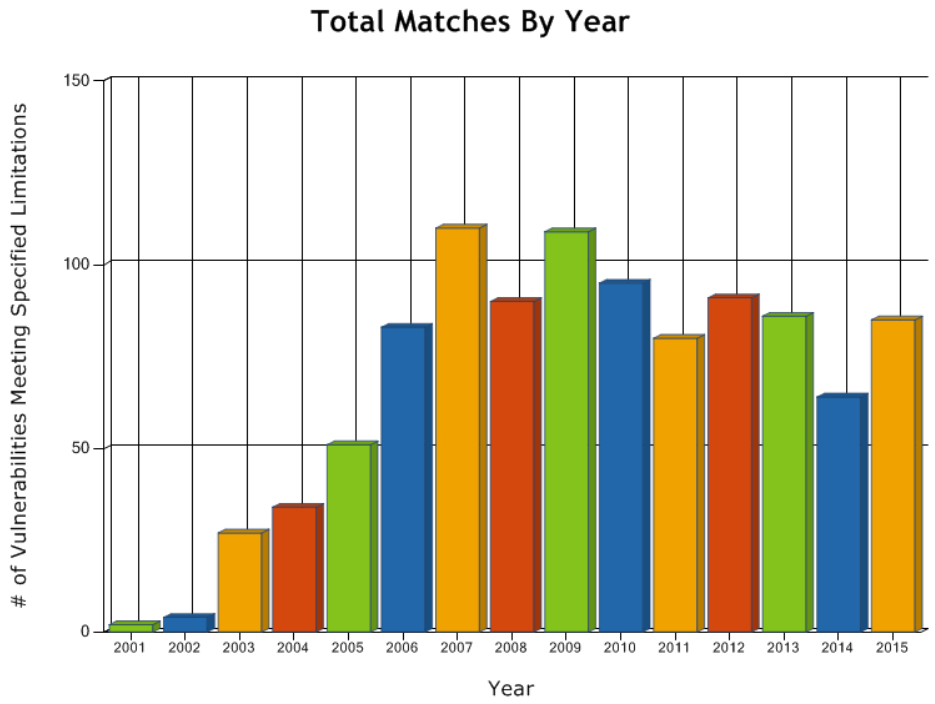
Some stats – Format string vulnerabilities

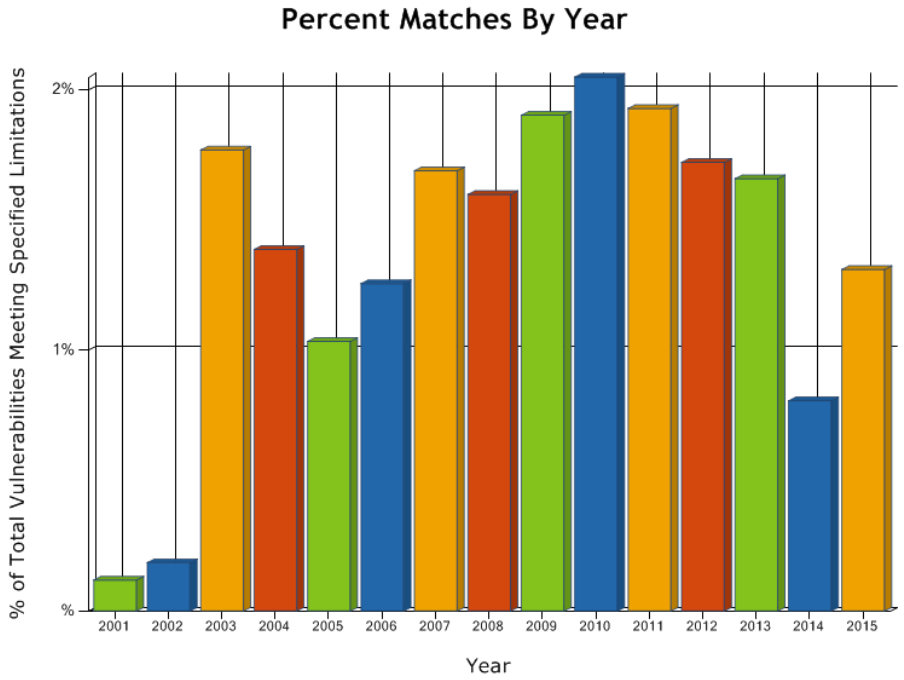




Source: NIST website

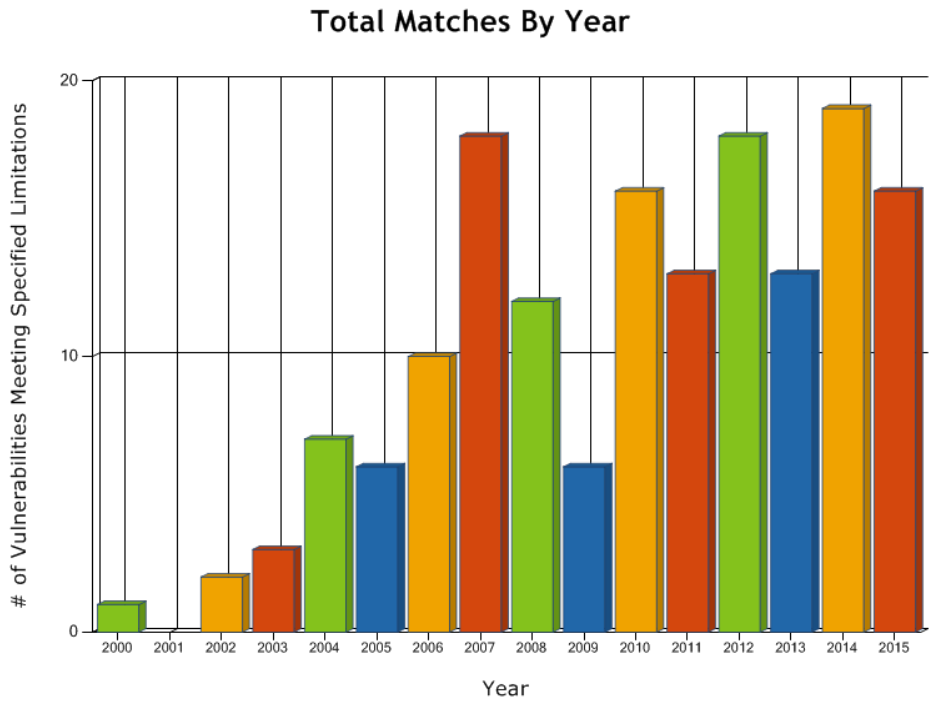
Some stats – Integer overflows

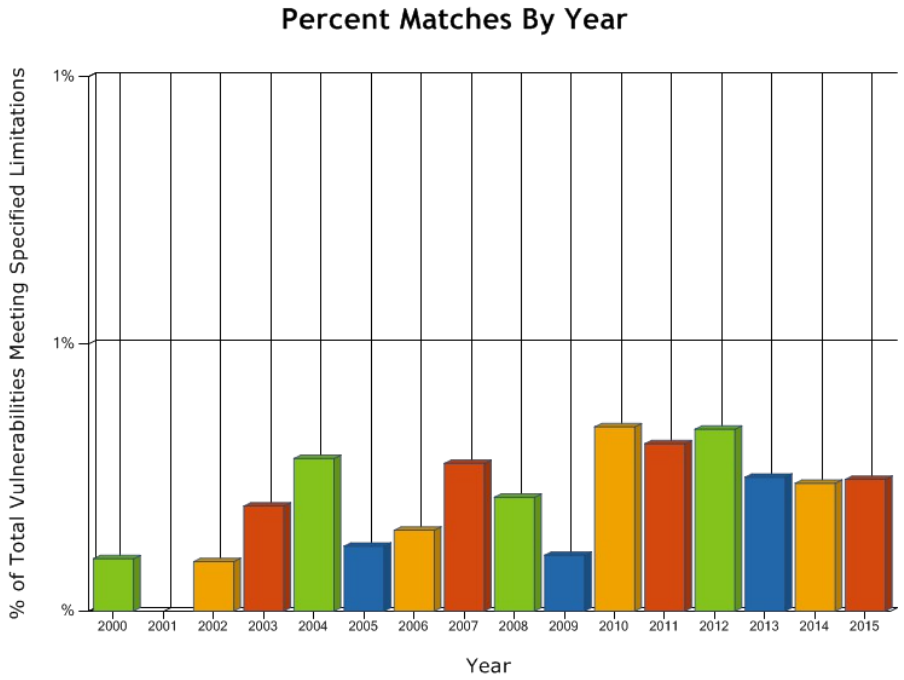




Source: NIST website

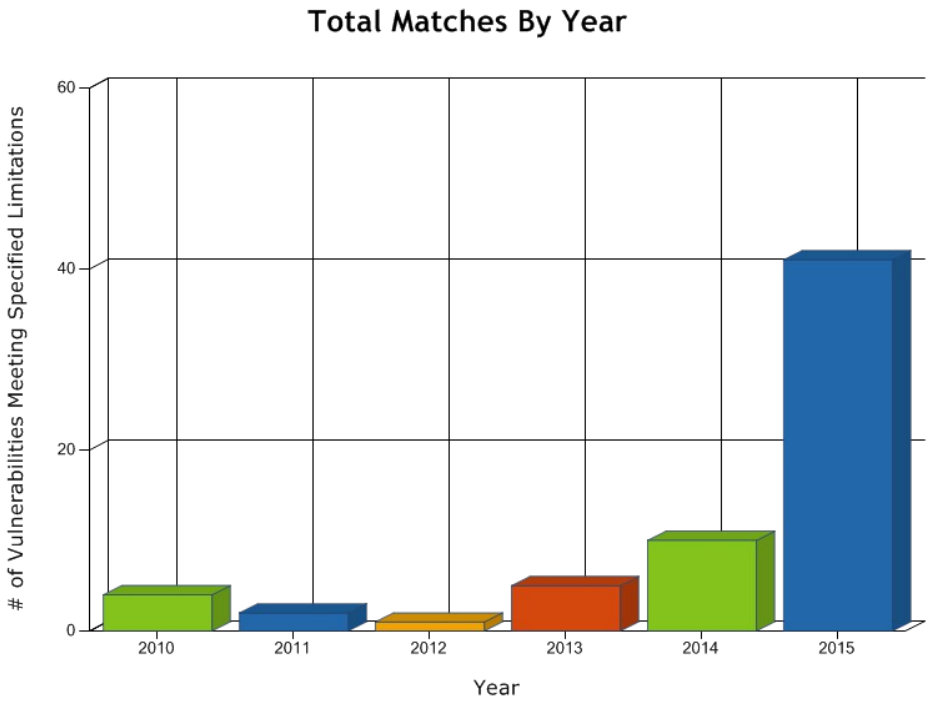
Some stats – Double free vulnerabilities

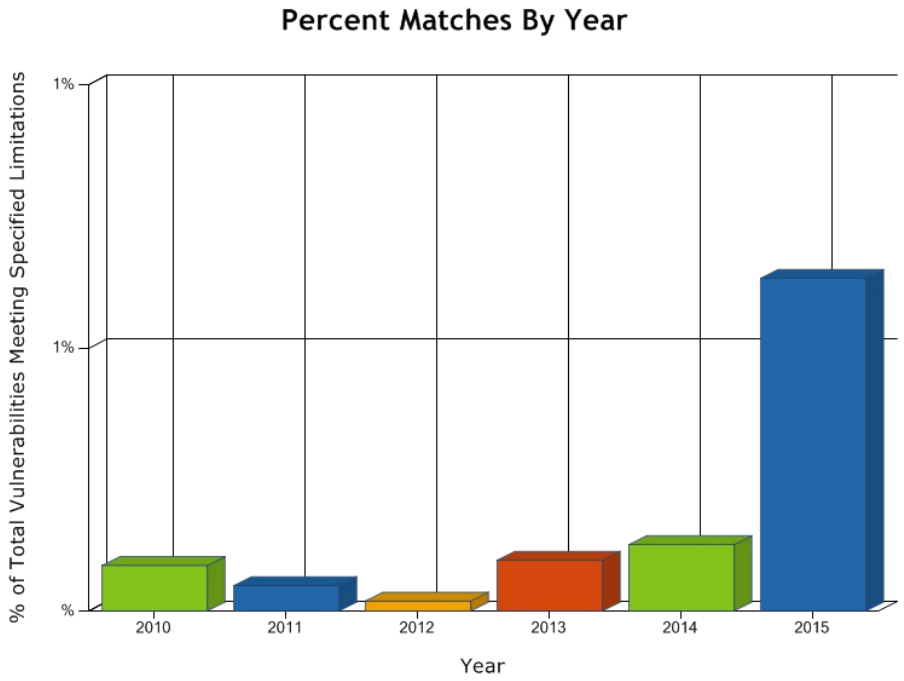




Source: NIST website

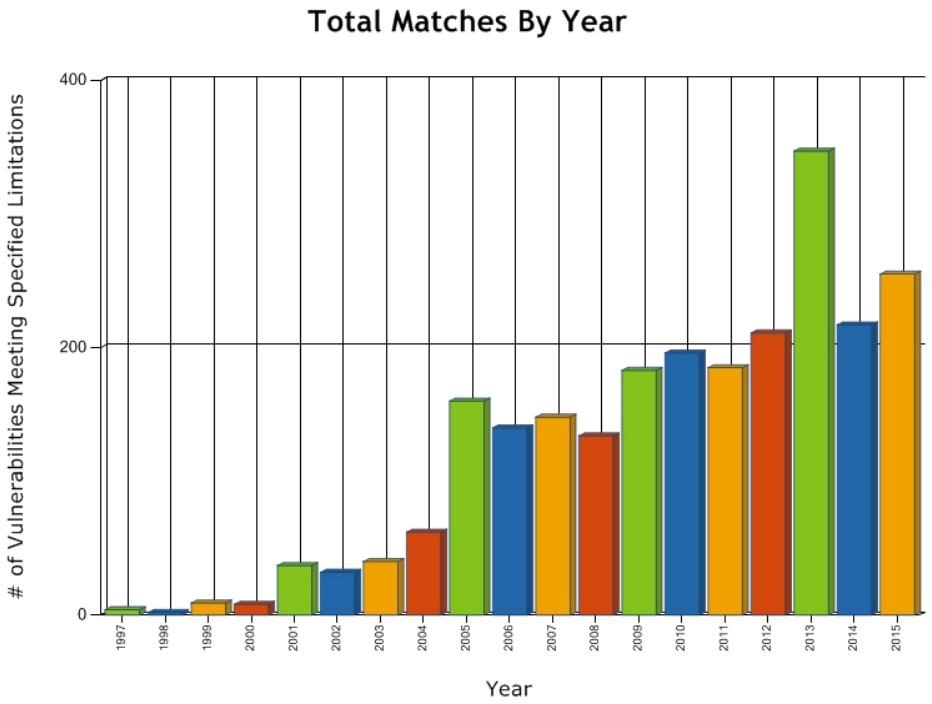
Some stats – Type confusion vulnerabilities

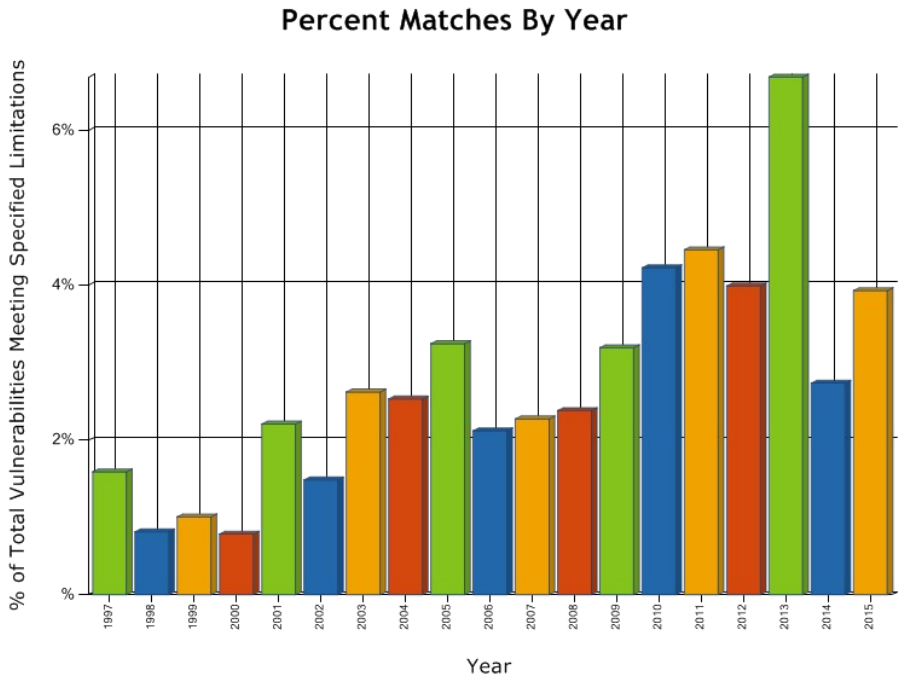




Source: NIST website

Some stats – Kernel vulnerabilities





Source: NIST website

Total Vulnerabilities by Operating System

Top operating systems by vulnerabilities reported in 2014

Operating system	# of vulnerabilities	# of HIGH vulnerabilities	# of MEDIUM vulnerabilities	# of LOW vulnerabilities
Apple Mac OS X	147	64	67	16
Apple iOS	127	32	72	23
Linux Kernel	119	24	74	21
Microsoft Windows Server 2008	38	26	12	0
Microsoft Windows 7	36	25	11	0
Microsoft Windows Server 2012	38	24	14	0
Microsoft Windows 8	36	24	12	0
Microsoft Windows 8.1	36	24	12	0
Microsoft Windows Vista	34	23	11	0
Microsoft Windows RT	30	22	8	0

Still, Windows is the most attacked operating system by cybercriminals and malware (as you will see next week)
(source: GTI vulnerability report 2014)

Top Vulnerabilities by Application

Top applications by vulnerabilities reported in 2014

Application	# of vulnerabilities	# of HIGH vulnerabilities	# of MEDIUM vulnerabilities	# of LOW vulnerabilities
Microsoft Internet Explorer	242	220	22	0
Google Chrome	124	86	38	0
Mozilla Firefox	117	57	57	3
Adobe Flash Player	76	65	11	0
Oracle Java	104	50	46	8
Mozilla Thunderbird	66	36	29	1
Mozilla Firefox ESR	61	35	25	1
Adobe Air	45	38	7	0
Apple TV	86	29	49	8
Adobe Reader	44	37	7	0
Adobe Acrobat	43	35	8	0
Mozilla SeaMonkey	63	28	34	1

(source: GTI vulnerability report 2014)

Some Readings

"Much ado about NULL: Exploiting a kernel NULL dereference"
https://blogs.oracle.com/ksplice/entry/much_ado_about_null_exploiting1

"Taking advantage if non-terminated buffers"

http://www.opennet.ru/base/sec/p56-0x0e.txt.html

"Kernel-mode exploits primer"

http://www.iseclab.org/projects/vifuzz/docs/exploit.pdf

"Linux Kernel can SLUB overflow"

https://jon.oberheide.org/blog/2010/09/10/linux-kernel-can-slub-overflow/

"Exploiting Stack Overflows in the Linux kernel"

http://www.exploit-db.com/wp-content/themes/exploit/docs/15634.pdf

February 29, 2016

Mobile Security

Computer Security 2 (GA02)

Emiliano De Cristofaro, Gianluca Stringhini

Thanks to Manuel Egele and Wil Robertson for some of the material

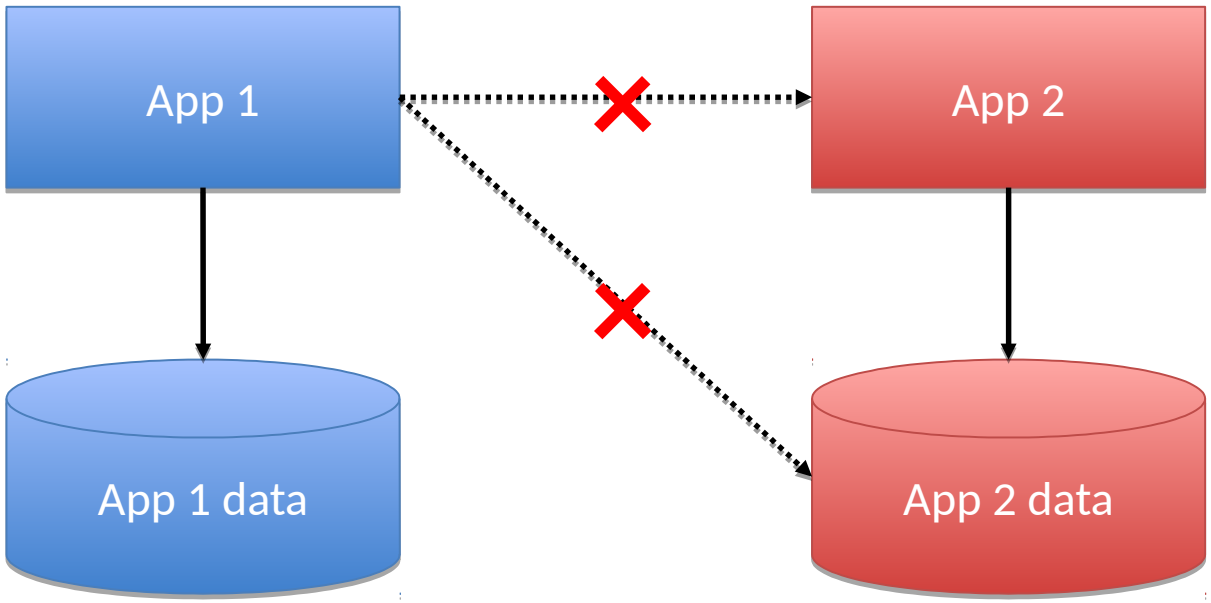
iOS Security Architecture

Privilege separation

- Built on UNIX security model
- Most processes run as the `mobile` user
- System daemons run under other least-privilege protection domains

Apps are siloed

- Little to no sharing outside of system apps



iOS Security Architecture

Memory protection

Data execution prevention (DEP), ASLR, sandboxing

ROP is still possible

Code signing framework

- All apps must be signed by Apple
- All apps must originate from the App Store (exception: **jailbreaking**, **provisioning**)

Curated app store model

- Apps places in the App Store are trusted
- Apps are “inspected” by Apple prior to publishing

iOS: Data protection API

- iOS provides an API for encrypting stored files
- Provides a number of protection classes
  1. File is protected, only accessible when device unlocked
  2. File is protected, accessible after device unlocked
  3. File is protected until user passcode entered
  4. File is not protected
- Keys derived from device UID key and user passcode
  - UID key resides in device crypto accelerator, never released

iOS: Provisioning

- All system apps must be signed by Apple's private key to execute
  - But, enterprises and universities often want to distribute their own apps outside of the App Store
- Provisioning provides these capabilities
  1. Apple signs certificates provided by developers
  2. Apple signs a provisioning profile that references developer certs
  3. Users install provisioning profile
  4. Device allows apps signed by developer's key to run according to the installed profile
- Similar process used for self-signed developer certs used during development

iOS: App review

- Central to iOS security is the curated App Store
  - Yes, Apple makes a nice profit by controlling the distribution channel
  - But, it also can have security benefits
- App review process gives Apple the chance to check for malicious submissions
- However, there is little transparency into what is actually done
  - Manual review by a small team
  - Checks for use of private APIs
  - Copyright, competition issues
  - Security?

InstaStock

- Charlie Miller submitted InstaStock to the App Store in 2011
  - Exploited dynamic code signing bug to download, map, and link arbitrary libraries at runtime
  - No obfuscation!
- The app passed the review process in a week
  - After publication, Charlie was kicked from the developer program

Android Security Architecture

Each app gets its own Java Virtual Machine instance and UNIX user ID

- App memory is physically separated, app data too
- Data Execution Prevention (DEP), ASLR
- Android apps are written in Java

Apps are sandboxed using a permission system

- Permissions declared in the app's manifest
- Users grant or deny permissions at install time
- Permissions map to UNIX groups (e.g., the camera group)

Problem: permissions are quite coarse-grained (e.g., Internet permission)

Android permissions

```
<manifest android:versionCode="1"
  android:versionName="1.0"
  package="com.someapp"
  xmlns:android="http://...">
  <uses-permission
    android:name=
      "android.permission.INTERNET"/>
  <uses-permission
    android:name=
      "android.permission.WRITE_INTERNAL_STORAGE"/>
  <uses-permission
    android:name=
      "android.permission.ACCESS_FINE_LOCATION"/>
</manifest>
```

Android Permissions

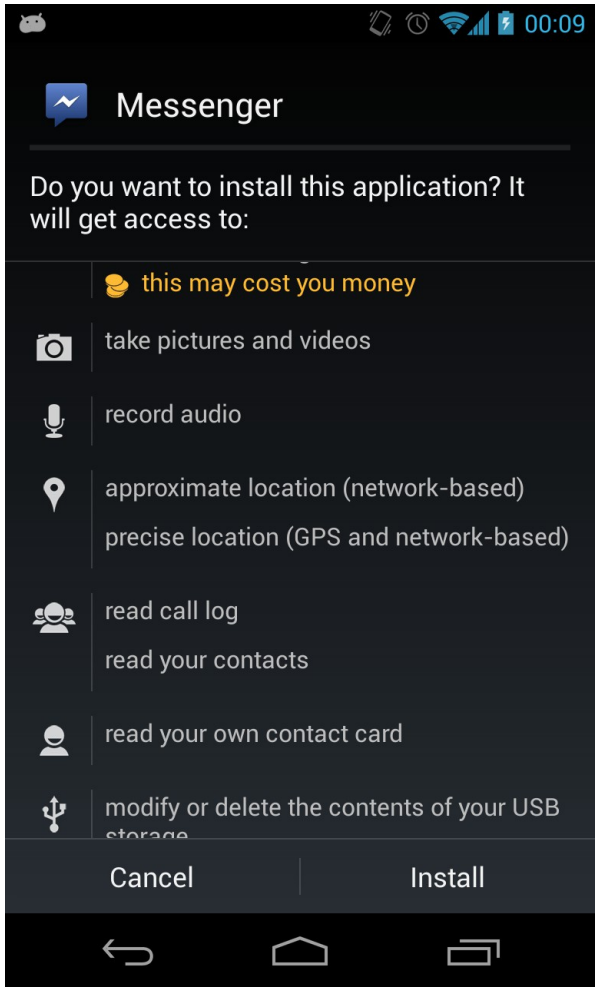


image:nelenkov.blogspot.com

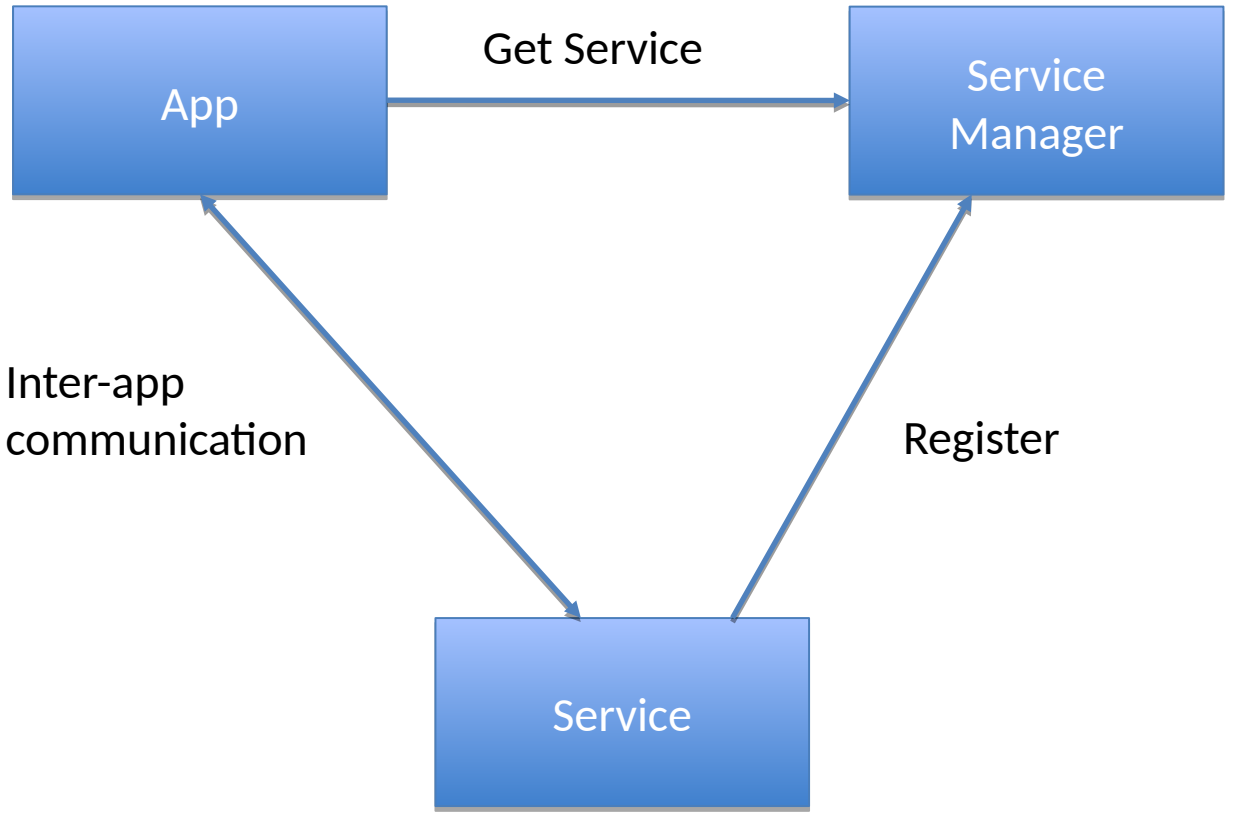
- **Install-time permissions** state up-front what an app is capable of doing
- No annoying popups during execution
- Are they effective?
- Studies have shown that only 20% of users even look at permissions

Android: Inter-app Communication

Android apps are sandboxed and run as separate users.

Inter-app communications is provided by a system service called **Binder**

All these messages go through the Binder service



In the paper "Man in the Binder: He who controls the IPC, controls the droid" the authors showed that attackers could hijack calls to the binder (for example by injecting libraries in apps) and steal data or tamper with it.

Android: Issues with Fragmentation

Unlike iOS, vendors are free to customise/manage their phones

This generates a fragmentation in the Android ecosystem, which has two main problems from the security side:

- Spotty system updates
- Use of unofficial app stores, opening them up to malware