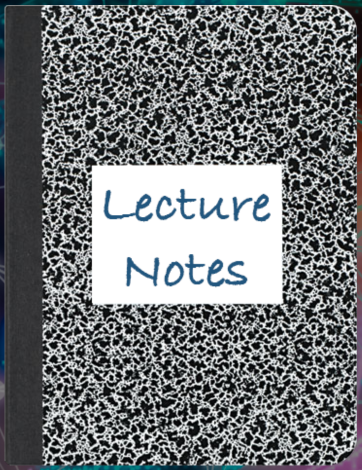
CS 419: Computer Security

Week 3: Code Injection



Paul Krzyzanowski

Part 1

Program Hijacking

Hijacking & Injection

Hijacking

Getting software to do something different from what the user or developer expected

- **Session hijacking**
  - Take over someone's communication session (typically from a web browser)
    - Usually involves stealing a session token that identifies the user and authorizes access
- **Program hijacking**
  - Get a program to execute unintended operations
  - **Command injection**
    - Send commands to a program that are then executed by the system shell
    - Includes SQL injection – send database commands
  - **Code injection**
    - Inject code into a program that is then executed by the application

Examples of Hijacking

- **Session hijacking**
  - Snoop on a communication session to get authentication info and take control of the session
- **Code injection**
  - Overflow an input buffer and cause new code to run
  - Provide JavaScript as input that will later get executed (Cross-site scripting)
  - Library injection: load different dynamic libraries that cause different versions of code run
- **Command injection**
  - Provide input that will get interpreted and run as a system command
  - Change search paths to run different programs
- **Other forms**
  - Redirect web browser to a malicious site
  - Change DNS (IP address lookup) results
  - Change search engine

Security-Sensitive Programs

- **Control hijacking isn't interesting for regular programs on your system**
  - You might as well just run commands from the shell
- **It *is* interesting if the program**
  - Has elevated privileges (*setuid*), especially runs as root
  - Runs on a system you don't have access to (most servers)

Privileged programs are more sensitive & more useful targets

Bugs and mistakes

- **Most attacks are due to**
  - Social engineering: getting a legitimate user to do something
  - Or bugs: using a program in a way it was not intended
    - Bugs include buggy security policies
- **Attacked system may be further weakened because of poor access control rules**
  - Violate Principle of Least Privilege
- **Cryptography won’t help us!**
  - And cryptographic software can also be buggy

Unchecked Assumptions

- **Unchecked assumptions can lead to vulnerabilities**
  - Vulnerability: weakness that can be exploited to perform unauthorized actions
- **Attack**
  - Discover assumptions
  - Craft an exploit to render them invalid … and run the exploit
- **Four common assumptions**
  1. Buffer is large enough for the data
  2. Integer overflow doesn't exist
  3. User input will never be processed as a command
  4. A file is in a proper format

Buffer Overflow

What is a buffer overflow?

Programming error that allows more data to be stored in an array than there is space

- **Buffer = stack, heap, or static data**
- **Overflow means adjacent memory will be overwritten**
  - Program data can be modified
  - New code can be injected
  - Unexpected transfer of control can be launched

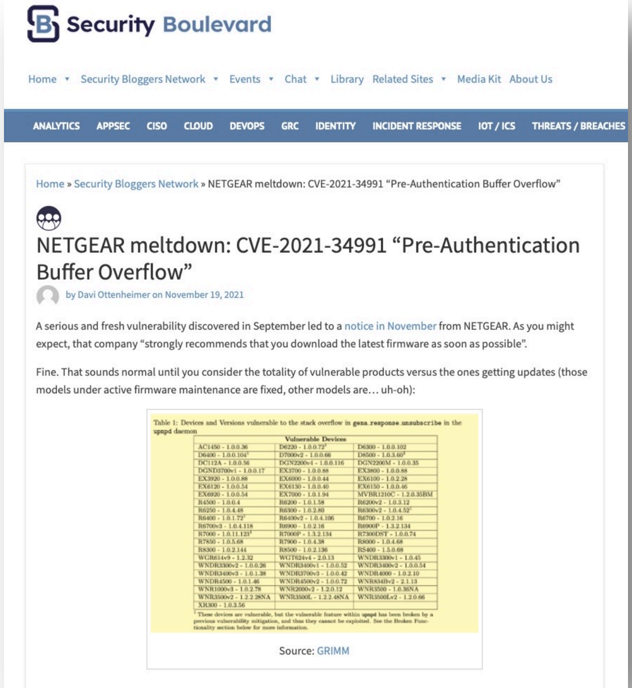
Buffer overflows

- **Buffer overflows used to be responsible for up to ~50% of vulnerabilities**
- **We know how to defend ourselves but**
  - Average time to patch a bug >> 1 year
  - People delay updating systems … or refuse to
  - Embedded systems often never get patched
    - Routers, cable modems, set-top boxes, access points, IP phones, and security cameras
  - Insecure access rights often help with gaining access or more privileges
  - We will continue to write buggy code!

Buffer overflows … still going strong

Nov. 19, 2021: NETGEAR meltdown

- Affects 61 different devices
- Allows attackers to execute arbitrary code on routers
- Authentication is not required for exploit
- Bug in UPnP service on TCP port 5000
  - When parsing the *uuid* request header, the process does not properly validate the length of user-supplied data prior to copying it to a fixed-length stack-based buffer.



Security Boulevard

Home • Security Bloggers Network • Events • Chat • Library Related Sites • Media Kit About Us

ANALYTICS APPSEC CISO CLOUD DEVOPS GRC IDENTITY INCIDENT RESPONSE IOT / ICS THREATS / BREACHES

Home » Security Bloggers Network » NETGEAR meltdown: CVE-2021-34991 "Pre-Authentication Buffer Overflow"

NETGEAR meltdown: CVE-2021-34991 "Pre-Authentication Buffer Overflow"

by Davi Ottenheimer on November 19, 2021

A serious and fresh vulnerability discovered in September led to a notice in November from NETGEAR. As you might expect, that company "strongly recommends that you download the latest firmware as soon as possible".

Fine. That sounds normal until you consider the totality of vulnerable products versus the ones getting updates (those models under active firmware maintenance are fixed, other models are… uh-oh):

Table 1: Devices and Versions vulnerable to the stack overflow in **gena_response_unsubscribe** in the **upnpd** daemon

Vulnerable Devices		
AC1450 - 1.0.0.36	D6220 - 1.0.0.72[1]	D6300 - 1.0.0.102
D6400 - 1.0.0.104[1]	D7000v2 - 1.0.0.66	D8500 - 1.0.3.60[1]
DC112A - 1.0.0.56	DGN2200v4 - 1.0.0.116	DGN2200M - 1.0.0.35
DGND3700v1 - 1.0.0.17	EX3700 - 1.0.0.88	EX3800 - 1.0.0.88
EX3920 - 1.0.0.88	EX6000 - 1.0.0.44	EX6100 - 1.0.2.28
EX6120 - 1.0.0.54	EX6130 - 1.0.0.40	EX6150 - 1.0.0.46
EX6920 - 1.0.0.54	EX7000 - 1.0.1.94	MVBR1210C - 1.2.0.35BM
R4500 - 1.0.0.4	R6200 - 1.0.1.58	R6200v2 - 1.0.3.12
R6250 - 1.0.4.48	R6300 - 1.0.2.80	R6300v2 - 1.0.4.52[1]
R6400 - 1.0.1.72[1]	R6400v2 - 1.0.4.106	R6700 - 1.0.2.16
R6700v3 - 1.0.4.118	R6900 - 1.0.2.16	R6900P - 1.3.2.134
R7000 - 1.0.11.123[1]	R7000P - 1.3.2.134	R7300DST - 1.0.0.74
R7850 - 1.0.5.68	R7900 - 1.0.4.38	R8000 - 1.0.4.68
R8300 - 1.0.2.144	R8500 - 1.0.2.136	RS400 - 1.5.0.68
WGR614v9 - 1.2.32	WGT624v4 - 2.0.13	WNDR3300v1 - 1.0.45
WNDR3300v2 - 1.0.0.26	WNDR3400v1 - 1.0.0.52	WNDR3400v2 - 1.0.0.54
WNDR3400v3 - 1.0.1.38	WNDR3700v3 - 1.0.0.42	WNDR4000 - 1.0.2.10
WNDR4500 - 1.0.1.46	WNDR4500v2 - 1.0.0.72	WNR834Bv2 - 2.1.13
WNR1000v3 - 1.0.2.78	WNR2000v2 - 1.2.0.12	WNR3500 - 1.0.36NA
WNR3500v2 - 1.2.2.28NA	WNR3500L - 1.2.2.48NA	WNR3500Lv2 - 1.2.0.66
XR300 - 1.0.3.56		

[1] These devices are vulnerable, but the vulnerable feature within **upnpd** has been broken by a previous vulnerability mitigation, and thus they cannot be exploited. See the Broken Functionality section below for more information.

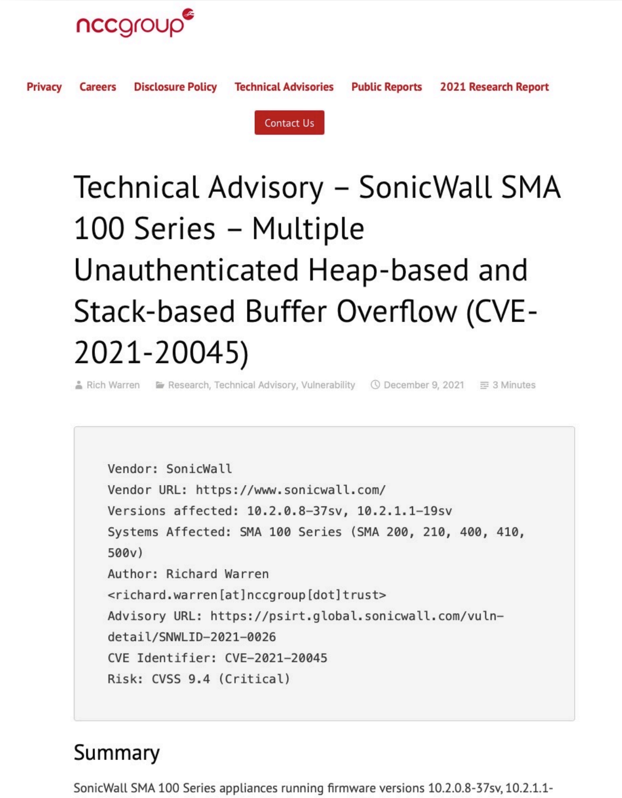
Source: GRIMM

https://securityboulevard.com/2021/11/netgear-meltdown-cve-2021-34991-pre-authentication-buffer-overflow/

Buffer overflows … still going strong

Dec. 9, 2021: SonicWall

- Affects SMA (Secure Mobile Access) 100 Series
- Multiple heap-based and stack-based buffer overflows
- Can be accessed by unauthenticated users
- Bug in `fileexplorer` component
  - Unchecked use of strcpy with a fixed size buffer
  - Assumes username and password will each be <128 bytes
  - Same bug with the domain name



nccgroup

Privacy | Careers | Disclosure Policy | Technical Advisories | Public Reports | 2021 Research Report

Contact Us

Technical Advisory – SonicWall SMA 100 Series – Multiple Unauthenticated Heap-based and Stack-based Buffer Overflow (CVE-2021-20045)

Rich Warren | Research, Technical Advisory, Vulnerability | December 9, 2021 | 3 Minutes

```
Vendor: SonicWall
Vendor URL: https://www.sonicwall.com/
Versions affected: 10.2.0.8-37sv, 10.2.1.1-19sv
Systems Affected: SMA 100 Series (SMA 200, 210, 400, 410, 500v)
Author: Richard Warren
<richard.warren[at]nccgroup[dot]trust>
Advisory URL: https://psirt.global.sonicwall.com/vuln-detail/SNWLID-2021-0026
CVE Identifier: CVE-2021-20045
Risk: CVSS 9.4 (Critical)
```

Summary

SonicWall SMA 100 Series appliances running firmware versions 10.2.0.8-37sv, 10.2.1.1-

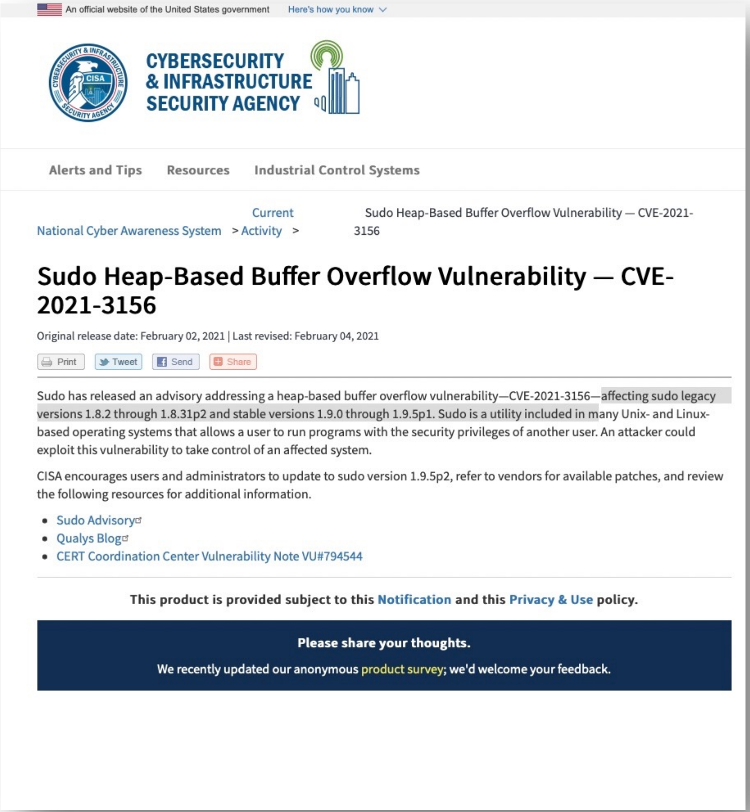
https://research.nccgroup.com/2021/12/09/technical-advisory-sonicwall-sma-100-series-multiple-unauthenticated-heap-based-and-stack-based-buffer-overflow-cve-2021-20045/

Buffer overflows … still going strong

Feb. 2, 2021: Linux sudo

- Heap-based buffer overflow vulnerability
- An attacker could exploit this vulnerability to take control of an affected system.

- Off-by-one error
  - Can result in a heap-based buffer overflow, which allows privilege escalation to root via "`sudoedit -s`" and a command-line argument that ends with a single backslash character.


An official website of the United States government Here's how you know

CYBERSECURITY & INFRASTRUCTURE SECURITY AGENCY

Alerts and Tips Resources Industrial Control Systems

National Cyber Awareness System > Current Activity > Sudo Heap-Based Buffer Overflow Vulnerability — CVE-2021-3156

Sudo Heap-Based Buffer Overflow Vulnerability — CVE-2021-3156

Original release date: February 02, 2021 | Last revised: February 04, 2021

Print Tweet Send Share

Sudo has released an advisory addressing a heap-based buffer overflow vulnerability—CVE-2021-3156—affecting sudo legacy versions 1.8.2 through 1.8.31p2 and stable versions 1.9.0 through 1.9.5p1. Sudo is a utility included in many Unix- and Linux-based operating systems that allows a user to run programs with the security privileges of another user. An attacker could exploit this vulnerability to take control of an affected system.

CISA encourages users and administrators to update to sudo version 1.9.5p2, refer to vendors for available patches, and review the following resources for additional information.

- Sudo Advisory
- Qualys Blog
- CERT Coordination Center Vulnerability Note VU#794544

This product is provided subject to this Notification and this Privacy & Use policy.

Please share your thoughts.

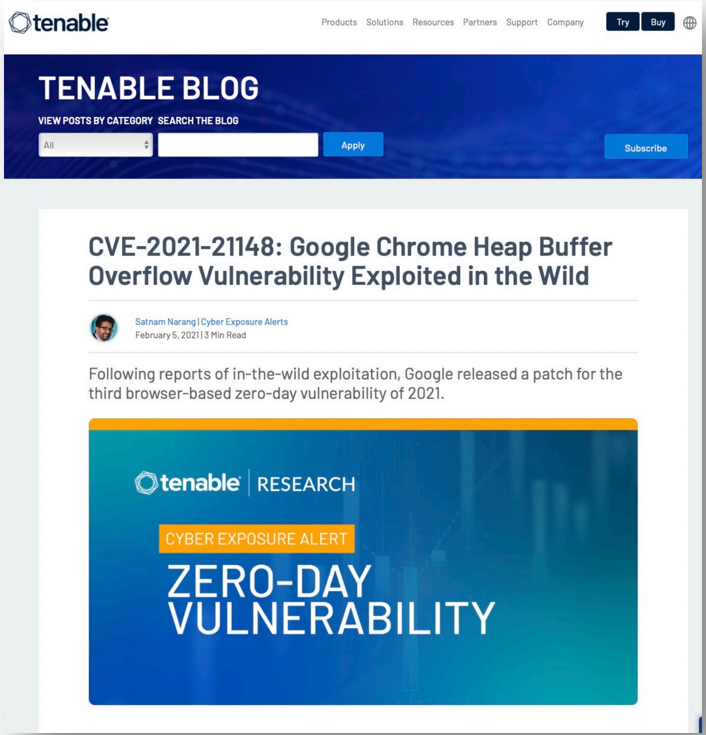
We recently updated our anonymous product survey; we'd welcome your feedback.

https://www.cisa.gov/uscert/ncas/current-activity/2021/02/02/sudo-heap-based-buffer-overflow-vulnerability-cve-2021-3156

Buffer overflows … still going strong

Feb. 5, 2021: Google Chrome

- Buffer overflow vulnerability in V8, Google Chrome's open-source JavaScript and WebAssembly engine
- Exploits in the wild have been observed
- Allows remote attacker to exploit heap corruption via a crafted HTML page
- Affects Microsoft Edge (Chromium based)

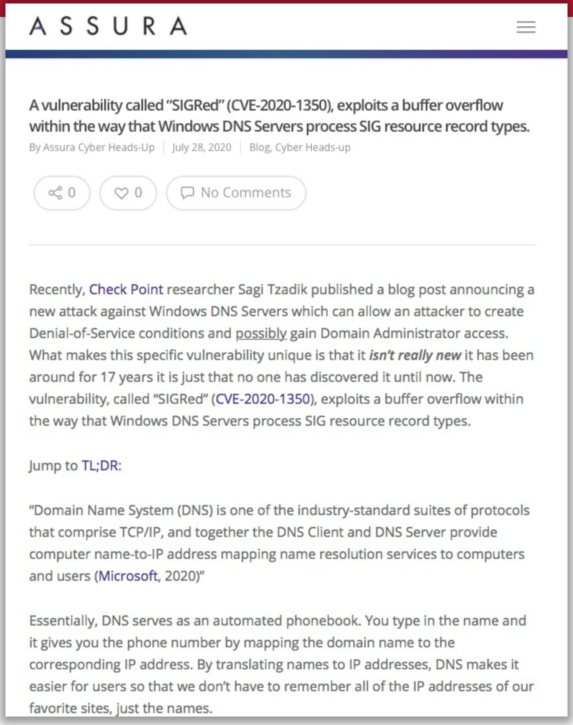


https://www.tenable.com/blog/cve-2021-21148-google-chrome-heap-buffer-overflow-vulnerability-exploited-in-the-wild

Buffer overflows … still going strong

July 28, 2020 – SIGRed vulnerability

– Exploits buffer overflow in Windows DNS Server processing of SIG records
– Allows an attacker to create a denial-of-service attack (& maybe get admin access)
– Bug existed for 17 years – discovered in 2020!
  - A function expects 16-bit integers to be passed to it
  - If they are not the proper size, it will overflow other integers
  - Attacker needs to create a DNS response that contains a SIG record > 64KB



ASSURA

A vulnerability called "SIGRed" (CVE-2020-1350), exploits a buffer overflow within the way that Windows DNS Servers process SIG resource record types.

By Assura Cyber Heads-Up | July 28, 2020 | Blog, Cyber Heads-up

0 0 No Comments

Recently, Check Point researcher Sagi Tzadik published a blog post announcing a new attack against Windows DNS Servers which can allow an attacker to create Denial-of-Service conditions and possibly gain Domain Administrator access. What makes this specific vulnerability unique is that it *isn't really new* it has been around for 17 years it is just that no one has discovered it until now. The vulnerability, called "SIGRed" (CVE-2020-1350), exploits a buffer overflow within the way that Windows DNS Servers process SIG resource record types.

Jump to TL;DR:

"Domain Name System (DNS) is one of the industry-standard suites of protocols that comprise TCP/IP, and together the DNS Client and DNS Server provide computer name-to-IP address mapping name resolution services to computers and users (Microsoft, 2020)"

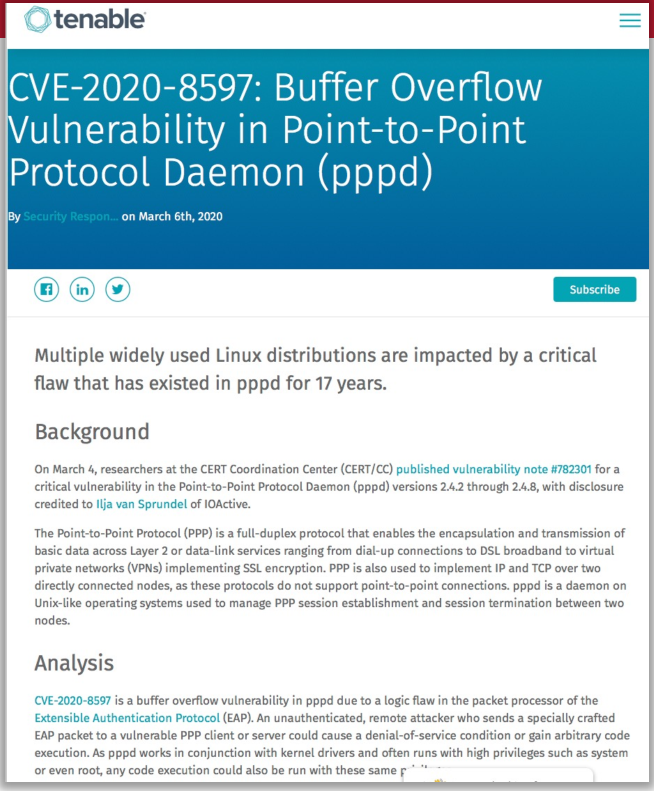
Essentially, DNS serves as an automated phonebook. You type in the name and it gives you the phone number by mapping the domain name to the corresponding IP address. By translating names to IP addresses, DNS makes it easier for users so that we don't have to remember all of the IP addresses of our favorite sites, just the names.

https://www.assurainc.com/a-vulnerability-called-sigred-cve-2020-1350-exploits-a-buffer-overflow-within-the-way-that-windows-dns-servers-process-sig-resource-record-types/amp-on/

Another 17 year-old bug

March 4, 2020: Point-to-Point Protocol Daemon

- *pppd* is used for layer 2 (data link) services that include DSL and VPNs
- Bug existed for 17 years – discovered in 2020!
  - Attacker creates a specially-crafted Extensible Authentication Protocol (EAP) message
  - Incorrect bounds check allows copying an arbitrary length of data



tenable

CVE-2020-8597: Buffer Overflow Vulnerability in Point-to-Point Protocol Daemon (pppd)

By Security Respon... on March 6th, 2020

Subscribe

Multiple widely used Linux distributions are impacted by a critical flaw that has existed in pppd for 17 years.

Background

On March 4, researchers at the CERT Coordination Center (CERT/CC) published vulnerability note #782301 for a critical vulnerability in the Point-to-Point Protocol Daemon (pppd) versions 2.4.2 through 2.4.8, with disclosure credited to Ilja van Sprundel of IOActive.

The Point-to-Point Protocol (PPP) is a full-duplex protocol that enables the encapsulation and transmission of basic data across Layer 2 or data-link services ranging from dial-up connections to DSL broadband to virtual private networks (VPNs) implementing SSL encryption. PPP is also used to implement IP and TCP over two directly connected nodes, as these protocols do not support point-to-point connections. pppd is a daemon on Unix-like operating systems used to manage PPP session establishment and session termination between two nodes.

Analysis

CVE-2020-8597 is a buffer overflow vulnerability in pppd due to a logic flaw in the packet processor of the Extensible Authentication Protocol (EAP). An unauthenticated, remote attacker who sends a specially crafted EAP packet to a vulnerable PPP client or server could cause a denial-of-service condition or gain arbitrary code execution. As pppd works in conjunction with kernel drivers and often runs with high privileges such as system or even root, any code execution could also be run with these same

https://www.tenable.com/blog/cve-2020-8597-buffer-overflow-vulnerability-in-point-to-point-protocol-daemon-pppd

GRUB2 Bootloader

July 29, 2020: GRUB2 bootloader

- Used by most Linux systems and many hypervisors and Windows systems that use Secure Boot with the standard Microsoft Third Party UEFI Certificate Authority
- Vulnerability allows attackers to gain arbitrary code execution during the boot process – even when Secure Boot is enabled
- Attacker needs to modify the GRUB2 config file
  - But this allows the attack to persist and launch new attacks even before the operating system boots
- GRUB2 checks a buffer size for a token
  - But does not quit if the token is too large

eclypsium

THERE'S A HOLE IN THE BOOT

"BootHole" vulnerability in the GRUB2 bootloader opens up Windows and Linux devices using Secure Boot to attack. All operating systems using GRUB2 with Secure Boot must release new installers and bootloaders.

Eclypsium researchers, Mickey Shkatov and Jesse Michael, have discovered a vulnerability – dubbed "BootHole" – in the GRUB2 bootloader utilized by most Linux systems that can be used to gain arbitrary code execution during the boot process, even when Secure Boot is enabled. Attackers exploiting this vulnerability can install persistent and stealthy bootkits or malicious bootloaders that could give them near-total control over the victim device.

The vulnerability affects systems using Secure Boot, even if they are not using GRUB2. Almost all signed versions of GRUB2 are vulnerable, meaning virtually every Linux distribution is affected. In addition, GRUB2 supports other operating systems, kernels and hypervisors such as Xen. The problem also extends to any Windows device that uses Secure Boot with the standard Microsoft Third Party UEFI Certificate Authority. Thus the majority of laptops, desktops, servers and workstations are affected, as well as network appliances and other special purpose equipment used in industrial, healthcare, financial and other industries. This vulnerability makes these devices susceptible to attackers such as the threat actors recently discovered using malicious UEFI bootloaders.

Eclypsium has coordinated the responsible disclosure of this vulnerability with a variety of industry entities, including OS vendors, computer manufacturers, and CERTs. Mitigation will require new bootloaders to be signed and deployed, and vulnerable bootloaders should be revoked to prevent adversaries from using older, vulnerable versions in an attack. This will likely be a long process and take considerable time for organizations to complete patching.

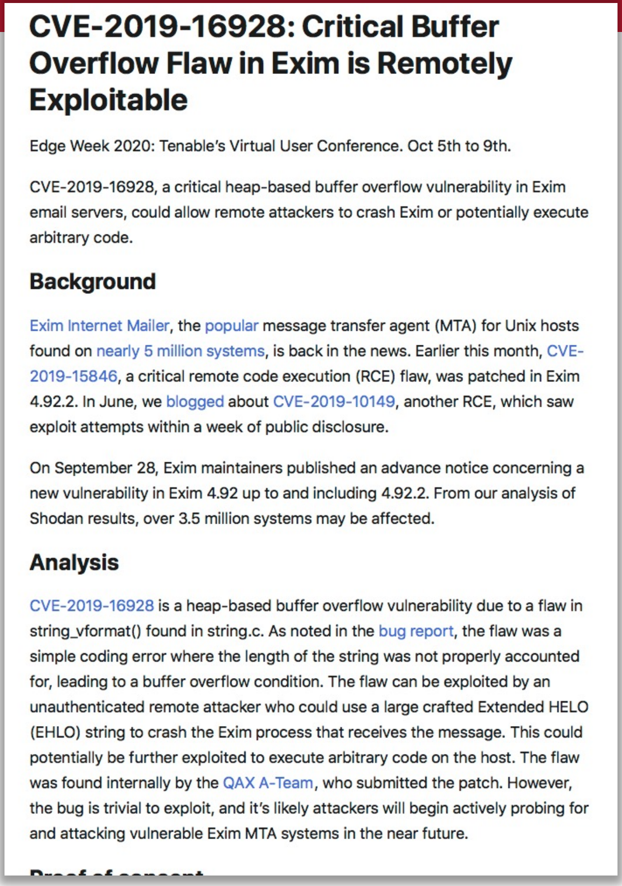
TABLE OF CONTENTS

https://eclypsium.com/wp-content/uploads/2020/08/Theres-a-Hole-in-the-Boot.pdf

Exim Mail Server Vulnerability

September 28, 2019: Exim server

- Heap-based buffer overflow vulnerability in Exim email
- Exim mail transfer agent used on 5 million systems
- Remote code execution possible because of a bug in `string_vformat()` found in `string.c`
- Length of the string was not properly accounted for



CVE-2019-16928: Critical Buffer Overflow Flaw in Exim is Remotely Exploitable

Edge Week 2020: Tenable's Virtual User Conference. Oct 5th to 9th.

CVE-2019-16928, a critical heap-based buffer overflow vulnerability in Exim email servers, could allow remote attackers to crash Exim or potentially execute arbitrary code.

Background

Exim Internet Mailer, the popular message transfer agent (MTA) for Unix hosts found on nearly 5 million systems, is back in the news. Earlier this month, CVE-2019-15846, a critical remote code execution (RCE) flaw, was patched in Exim 4.92.2. In June, we blogged about CVE-2019-10149, another RCE, which saw exploit attempts within a week of public disclosure.

On September 28, Exim maintainers published an advance notice concerning a new vulnerability in Exim 4.92 up to and including 4.92.2. From our analysis of Shodan results, over 3.5 million systems may be affected.

Analysis

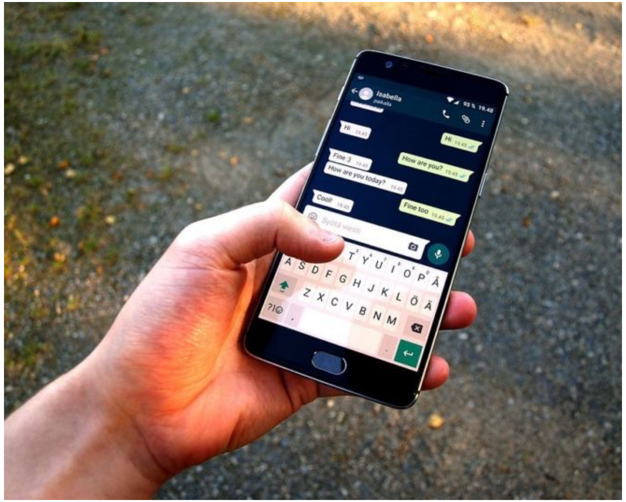
CVE-2019-16928 is a heap-based buffer overflow vulnerability due to a flaw in string_vformat() found in string.c. As noted in the bug report, the flaw was a simple coding error where the length of the string was not properly accounted for, leading to a buffer overflow condition. The flaw can be exploited by an unauthenticated remote attacker who could use a large crafted Extended HELO (EHLO) string to crash the Exim process that receives the message. This could potentially be further exploited to execute arbitrary code on the host. The flaw was found internally by the QAX A-Team, who submitted the patch. However, the bug is trivial to exploit, and it's likely attackers will begin actively probing for and attacking vulnerable Exim MTA systems in the near future.

ars technica

WhatsApp vulnerability exploited to infect phones with Israeli spyware

Attacks used app's call function. Targets didn't have to answer to be infected.

DAN GOODIN - 5/13/2019, 10:00 PM



Attackers have been exploiting a vulnerability in WhatsApp that allowed them to infect phones with advanced spyware made by Israeli developer NSO Group, the Financial Times reported on Monday, citing the company and a spyware technology dealer.

A representative of WhatsApp, which is used by 1.5 billion people, told Ars that company researchers discovered the vulnerability earlier this month while they were making security improvements. CVE-2019-3568, as the vulnerability has been indexed, is a buffer overflow vulnerability in the WhatsApp VOIP stack that allows remote code execution when specially crafted series of SRTCP packets are sent to a target phone number, according to this advisory.

https://arstechnica.com/information-technology/2019/05/whatsapp-vulnerability-exploited-to-infect-phones-with-israeli-spyware/

2019 WhatsApp Buffer Overflow Vulnerability

- **WhatsApp messaging app could install malware on Android, iOS, Windows, & Tizen operating systems**

  An attacker did not have to get the user to do anything: the attacker just places a WhatsApp voice call to the victim ⇒ **zero-click attack**

- **This was a zero-day vulnerability**
  - Attackers found & exploited the bug before the company could patch it

- **WhatsApp used by 1.5 billion people**
  - Vulnerability discovered in May 2019 while developers were making security improvements

https://arstechnica.com/information-technology/2019/05/whatsapp-vulnerability-exploited-to-infect-phones-with-israeli-spyware/

Many, many more!

CVE-2021-45526 Certain NETGEAR devices are affected by a buffer overflow by an authenticated user. This affects EX6000 before 1.0.0.38, EX6120 before 1.0.0.48, EX6130 before 1.0.0.30, R6300v2 before 1.0.4.52, R6400 before 1.0.1.52, R7000 before 1.0.11.126, R7900 before 1.0.4.30, R8000 before 1.0.4.52, R7000P before 1.3.2.124, R8000P before 1.4.1.50, RAX80 before 1.0.3.88, R6900P before 1.3.2.124, R7900P before 1.4.1.50, and RAX75 before 1.0.3.88.

CVE-2021-45525 Certain NETGEAR devices are affected by a buffer overflow by an authenticated user. This affects EX7000 before 1.0.1.80, R6400 before 1.0.1.50, R6400v2 before 1.0.4.118, R6700 before 1.0.2.8, R6700v3 before 1.0.4.118, R6900 before 1.0.2.8, R6900P before 1.3.2.124, R7000 before 1.0.9.88, R7000P before 1.3.2.124, R7900 before 1.0.3.18, R7900P before 1.4.1.50, R8000 before 1.0.4.46, R8000P before 1.4.1.50, RAX80 before 1.0.1.56, and WNR3500Lv2 before 1.2.0.62.

CVE-2021-45524 NETGEAR R8000 devices before 1.0.4.62 are affected by a buffer overflow by an authenticated user.

CVE-2021-45523 NETGEAR R7000 devices before 1.0.9.42 are affected by a buffer overflow by an authenticated user.

CVE-2021-45417 AIDE before 0.17.4 allows local users to obtain root privileges via crafted file metadata (such as XFS extended attributes or tmpfs ACLs), because of a heap-based buffer overflow.

CVE-2021-45342 A buffer overflow vulnerability in CDataList of the jwwlib component of LibreCAD 2.2.0-rc3 and older allows an attacker to achieve Remote Code Execution using a crafted JWW document.

CVE-2021-45341 A buffer overflow vulnerability in CDataMoji of the jwwlib component of LibreCAD 2.2.0-rc3 and older allows an attacker to achieve Remote Code Execution using a crafted JWW document.

CVE-2021-45078 stab_xcoff_builtin_type in stabs.c in GNU Binutils through 2.37 allows attackers to cause a denial of service (heap-based buffer overflow) or possibly have unspecified other impact, as demonstrated by an out-of-bounds write. NOTE: this issue exists because of an incorrect fix for CVE-2018-12699.

CVE-2021-44847 A stack-based buffer overflow ... ttackers to crash the process or potentially execute arbitrar...

CVE-2021-44790 A carefully crafted request ... aft one. This issue affects Apache HTTP Server 2.4.51 and e...

CVE-2021-44738 Buffer overflow vulnerabil...

CVE-2021-44703 Acrobat Reader DC versio... oitrary code execution in the context of the current user. Exploi...

CVE-2021-44648 GNOME gdk-pixbuf 2.42.6...

CVE-2021-44538 The olm_session_describe... e party of the channel. Attackers can construct a crafted se... undocumented. The overflow content is partially contro...

CVE-2021-44435 A vulnerability has been i... could leverage this vulnerability to execute code in the conte...

CVE-2021-44432 A vulnerability has been i... could leverage this vulnerability to execute code in the conte...

CVE-2021-44422 An Improper Input Validat... heap-based buffer overflow. An attacker can leverage this...

CVE-2021-44352 A Stack-based Buffer Ove...

CVE-2021-44165 A vulnerability has been i... < V2.41). The affected firmware contains a buffer overflow...

CVE-2021-44158 ASUS RT-AX56U Wi-Fi Route... or disrupt service.

CVE-2021-44154 An issue was discovered in Rep... a buffer overflow.

522 reported buffer overflow vulnerabilities
Jan 6 2021 – Feb 7, 2022

CVE-2021-43983 WECON LeviStudioU Versions 2019-09-21 and prior are vulnerable to multiple stack-based buffer overflow instances while parsing project files, which may allow an attacker to execute arbitrary code.

CVE-2021-43982 Delta Electronics CNCSoft Versions 1.01.30 and prior are vulnerable to a stack-based buffer overflow, which may allow an attacker to execute arbitrary code.

CVE-2021-43637 Amazon WorkSpaces agent is affected by Buffer Overflow. IOCTL Handler 0x22001B in the Amazon WorkSpaces agent below v1.0.1.1537 allow local attackers to execute arbitrary code in kernel mode or cause a denial of service (memory corruption and OS crash) via specially crafted I/O Request Packet.

CVE-2021-43618 GNU Multiple Precision Arithmetic Library (GMP) through 6.2.1 has an mpz/inp_raw.c integer overflow and resultant buffer overflow via crafted input, leading to a segmentation fault on 32-bit platforms.

CVE-2021-43579 A stack-based buffer overflow in image_load_bmp() in HTMLDOC <= 1.9.13 results in remote code execution if the victim converts an HTML document linking to a crafted BMP file.

CVE-2021-43573 A buffer overflow was discovered on Realtek RTL8195AM devices before 2.0.10. It exists in the client code when processing a malformed IE length of HT capability information in the Beacon and Association response frame.

CVE-2021-43556 FATEK WinProladder Versions 3.30_24518 and prior are vulnerable to a stack-based buffer overflow while processing project files, which may allow an attacker to execute arbitrary code.

CVE-2021-43518 Teeworlds up to and including 0.7.5 is vulnerable to Buffer Overflow. A map parser does not validate m_Channels value coming from a map file, leading to a buffer overflow. A malicious server may offer a specially crafted map that will overwrite client's stack causing denial of service or code execution.

CVE-2021-43280 A stack-based buffer overflow vulnerability exists in the DWF file reading procedure in Open Design Alliance Drawings SDK before 2022.8. The issue ... n attacker can leverage this vulnerability to execute code in the context of the current process.

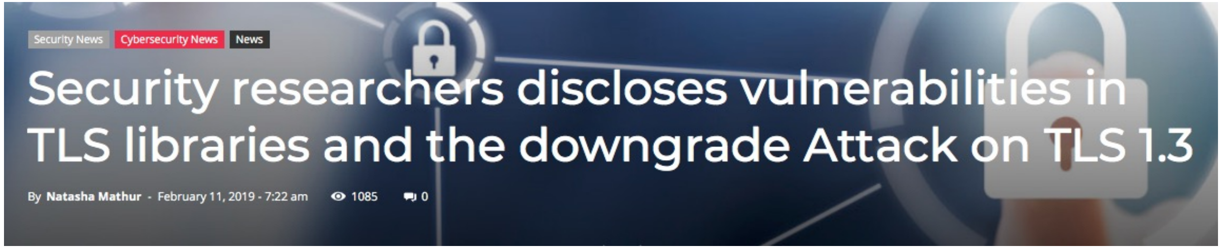
https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow

Buggy libraries can affect a lot of code bases

July 2017 – Devil's Ivy (CVE-2017-9765)

- gsoap open source toolkit
- Enables remote attacker to execute arbitrary code
- Discovered during the analysis of an internet-connected security camera

> **Millions of IoT devices are vulnerable to buffer overflow attack**
>
> July 18, 2017 · Eslam Medhat · 104 Views · 0 Comments · buffer overflow
>
> A buffer overflow flaw has been found by security researchers (at the IoT-focused security firm Senrio) in an open-source software development library that is widely used by major manufacturers of the Internet-of-Thing devices.
>
> The buffer overflow vulnerability (CVE-2017-9765), which is called "Devil's Ivy" enables a remote attacker to crash the SOAP (Simple Object Access Protocol) WebServices daemon and make it possible to execute arbitrary code on the affected devices.



https://latesthackingnews.com/2017/07/18/millions-of-iot-devices-are-vulnerable-to-buffer-overflow-attack/

The classic buffer overflow bug

gets.c from OS X: © 1990,1992 The Regents of the University of California.

```
gets(buf)
char *buf;
  register char *s;
  static int warned;
  static char w[] = "warning: this program uses gets(),
  which is unsafe.\r\n";

  if (!warned) {
      (void) write(STDERR_FILENO, w, sizeof(w) - 1);
      warned = 1;
  }
  for (s = buf; (c = getchar()) != '\n';)
      if (c == EOF)
          if (s == buf)
              return (NULL);
          else
              break;
      else
          *s++ = c;
  *s = 0;
  return (buf);
}
```

```
...
char name[128];    /* user’s name */
...
printf("enter your name: ");
if (gets(name) != NULL)
  printf("your name is \"%s\"\n", name);
```

The classic buffer overflow bug

gets.c from OS X: © 1990,1992 The Regents of the University of California.

```
gets(buf)
char *buf;
  register char *s;
  static int warned;
  static char w[] = "warning: this program uses gets(),
  which is unsafe.\r\n";

  if (!warned) {
      (void) write(STDERR_FILENO, w, sizeof(w) - 1);
      warned = 1;
  }
  for (s = buf; (c = getchar()) != '\n';)
      if (c == EOF)
            if (s == buf)
                  return (NULL);
            else
                  break;
      else
            *s++ = c;
  *s = 0;
  return (buf);
}
```

gets.c from OS X: © 1990,1992 The Regents of the University of California.

```
gets(buf)
char *buf;
  register char *s;
  static int warned;
```

T

```
for (s = buf; (c = getchar()) != '\n';)
  if (c == EOF)
    if (s == buf)
      return (NULL);
    else
      break;
  else
    *s++ = c;
```

gets(),

1);

```
  *s = 0;
  return (buf);
}
```

C++ too – and no warnings!

```
#include <iostream>

using namespace std;

int main()
{
    char x[4] = "cat";
    char y[4];
    char z[4] = "dog";

    cout << "Enter a word:";
    cin >> y;
    cout << "Read " << strlen(y) << " characters." << endl;
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    cout << "z: " << z << endl;
}
```

C++ too – and no warnings!

```
#include <iostream>

using names

int main()
{
     char x[
     char y[
     char z[

     cout <<
     cin >>
     cout <<
     cout <<
     cout <<
     cout <<
}
```

```
$ g++ -o cin cin.cpp
Enter a word:abcdefg
Read 7 characters.
x: efg
y: abcdefg
z: dog
```

The data in `y` overflowed to `x`
`x` got corrupted

C++ too – and no warnings!

```
#include <iostream>

using names

int main()
{
      char x[
      char y[
      char z[

      cout <<
      cin >>
      cout <<
      cout <<
      cout <<
      cout <<
}
```

```
$ g++ -o cin cin.cpp
Enter a word:abcdefghijklmnopqrstuvwxyz0123456789
Read 36 characters.
x: efghijklmnopqrstuvwxyz0123456789
y: abcdefghijklmnopqrstuvwxyz0123456789
z: dog
Bus error: 10
```

With even more data,
`x` got corrupted
AND the program crashed!

Buffer overflow examples

```
void test(void) {
    char name[10];

    strcpy(name, "krzyzanowski");
}
```

That's easy to spot!

Another example

How about this?

```
char configfile[256];
char *base = getenv("BASEDIR");

if (base != NULL)
    sprintf(configfile, "%s/config.txt", base);
else {
    fprintf(stderr, "BASEDIR not set\n");
}
```

Buffer overflow attacks

To exploit a buffer overflow

- **Identify overflow vulnerability in a program**
  - Black box testing
    - Trial and error
    - Fuzzing tools (more on that …)
  - Inspection
    - Study the source
    - Trace program execution
- **Understand where the buffer is in memory and whether there is potential for corrupting surrounding data**

What's the harm?

Execute arbitrary code, such as starting a shell

Code injection, *stack smashing*

- Code runs with the privileges of the program
  - If the program is *setuid root* then you have root privileges
  - If the program is on a server, you can run code on that server

- **Even if you cannot execute code…**
  - You may crash the program or change how it behaves
  - Modify data
  - Denial of service attack

- **Sometimes the crashed code can leave a core dump**
  - You can access that and grab data the program had in memory

Taking advantage of unchecked bounds

```
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>

int
main(int argc, char **argv)
{
	char pass[5];
	int correct = 0;

	printf("enter password: ");
	gets(pass);
	if (strcmp(pass, "test") == 0) {
		printf("password is correct\n");
		correct = 1;
	}
	if (correct) {
		printf("authorized: running with root privileges...\n");
		exit(0);
	}
	else
		printf("sorry - exiting\n");
	exit(1);
}
```

```
$ ./buf
enter password: abcdefghijklmnop
authorized: running with root privileges...
```

Run on my Raspberry Pi
Raspbian GNU/Linux 10
5.10.63-v7l+
Or my Mac Mini M1 running macOS 12.2
Or my Intel i7 iMac running macOS 12.2

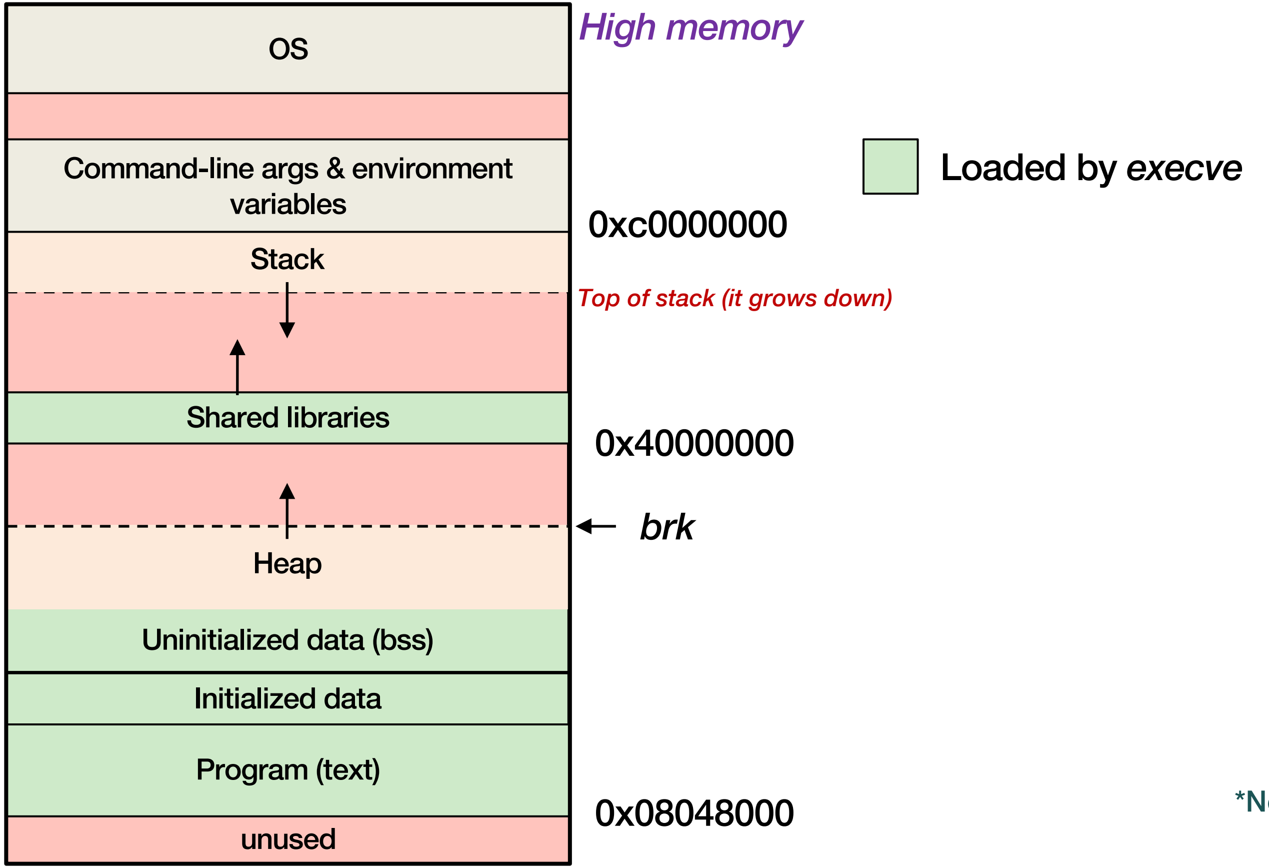
It’s a bounds checking problem

- **C and C++**
  - Allow direct access to memory
  - Do not check array bounds
  - Functions often do not even know array bounds
    - They just get passed a pointer to the start of an array
- **This is not a problem with strongly typed languages**
  - Java, C#, Python, etc. check sizes of structures
- **But C is in the top 4-5 of popular programming languages**
  - #1 for system programming & embedded systems
  - And most compilers, interpreters, and libraries are written in C

Part 2

Anatomy of overflows

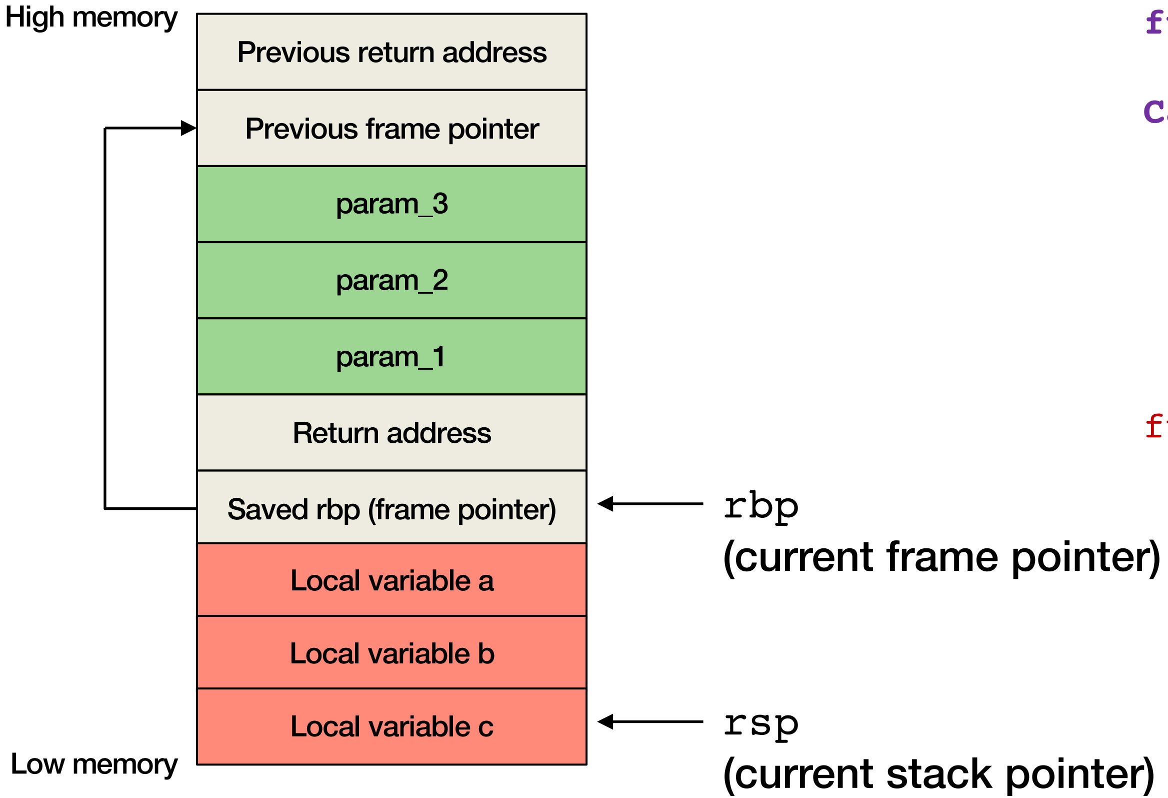
Linux process memory map*



*Not to scale

The stack

Note: `rbp` & `rsp` are used in 64-bit processors
`ebp` & `esp` are used in 32-bit processors




```
func(param_1, param_2, param_3)

Calling function:
          pushl param_3
          pushl param_2
          pushl param_1
          call func
          . . .

func:     pushl rbp
          movl %rsp, %rbp
          subl $20, %rsp
          . . .
          leave
          ret
```

Causing overflow

Overflow can occur when programs do not validate the length of data being written to a buffer

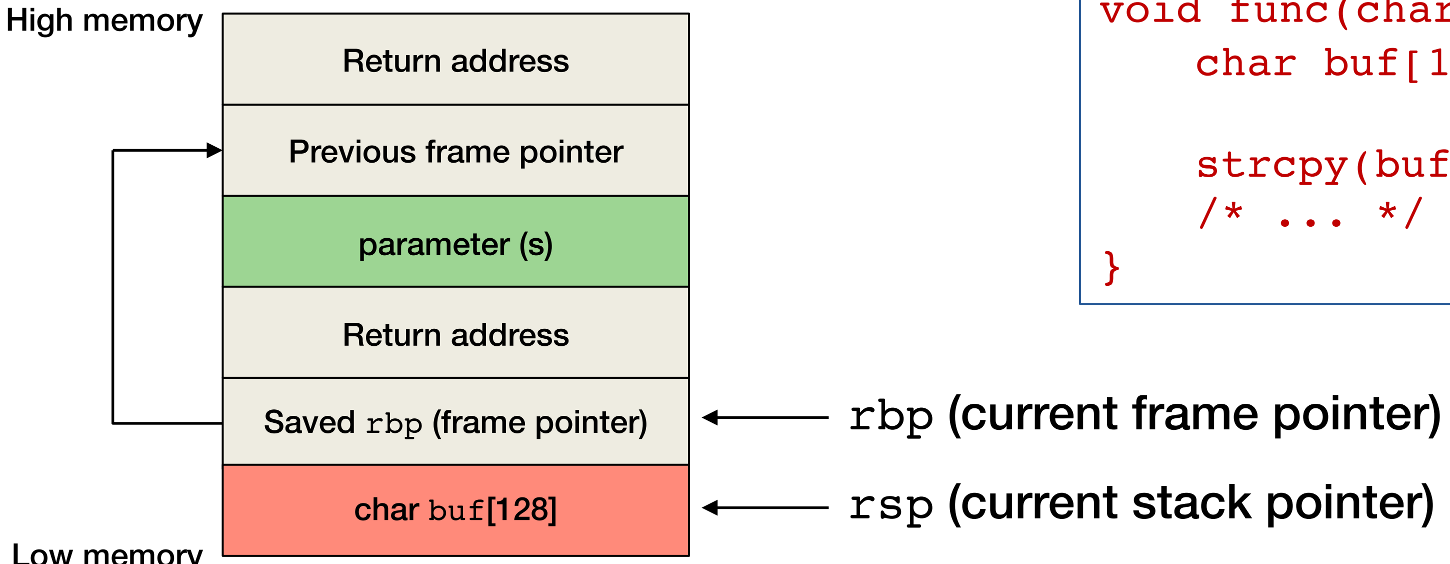
This could be in your code or one of several "unsafe" libraries

- `strcpy(char *dest, const char *src);`
- `strcat(char *dest, const char *src);`
- `gets(char *s);`
- `scanf(const char *format, …)`
- Others…

Overflowing the buffer

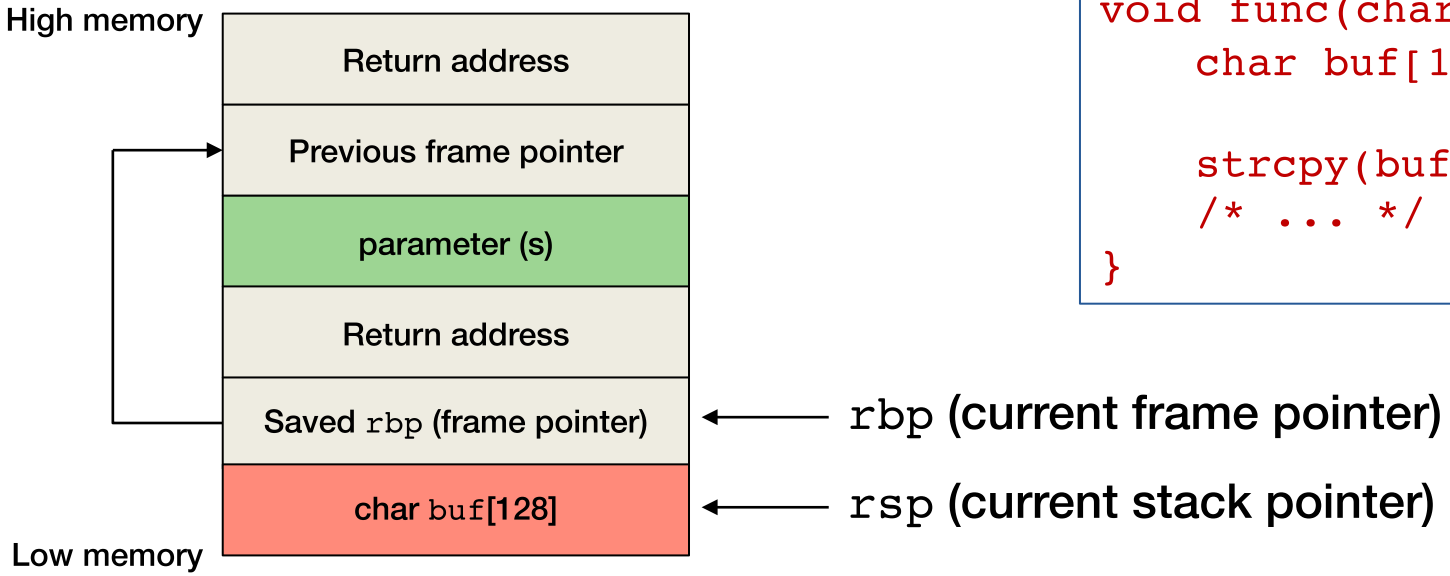
```
void func(char *s) {
    char buf[128];

    strcpy(buf, s);
    /* ... */
}
```



What if `strlen(s)` is >127 bytes?

Overflowing the buffer




```
void func(char *s) {
    char buf[128];

    strcpy(buf, s);
    /* ... */
}
```

What if `strlen(s)` is >127 bytes?

You overwrite the saved *rbp* and then the *return address*

Overwriting the return address

- **If we overwrite the return address**
  - We change what the program executes when it returns from the function
- **“Benign” overflow**
  - Overflow with garbage data
  - Chances are that the return address will be invalid
  - Program will die with a SEGFAULT
  - **Availability attack**

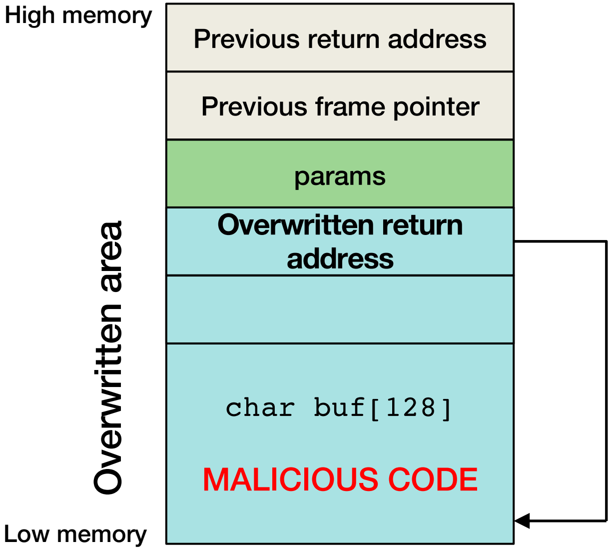
Programming at the machine level

- **High level languages (even C) constrain you in**
  - Access to variables (local vs. global)
  - Control flows in predictable ways
    - Loops, function entry/exit, exceptions
- **At the machine code level**
  - No restriction on where you can jump
    - Jump to the middle of a function … or to the middle of a C statement
    - Returns will go to whatever address is on the stack
    - Unused code can be executed (e.g., library functions you don’t use)

Subverting control flow

Malicious overflow

- Fill the buffer with malicious code
- Overflow to overwrite saved `%rbp`
- Then overwrite saved the `%rsp` (return address) with the address of the malicious code in the buffer

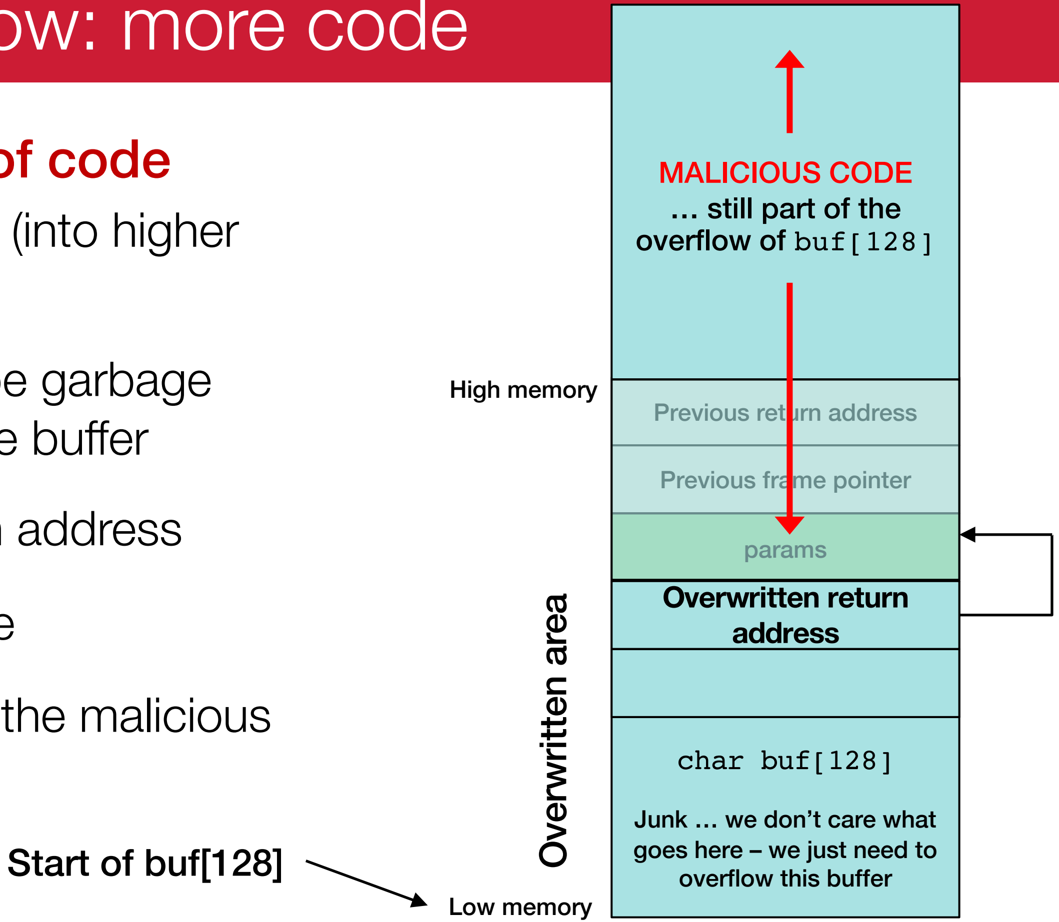


Subverting control flow: more code

If you want to inject a lot of code

Just go further down the stack (into higher memory

- Initial parts of the buffer will be garbage data … we just need to fill the buffer
- Then we have the new return address
- Then we have malicious code
- The return address points to the malicious code

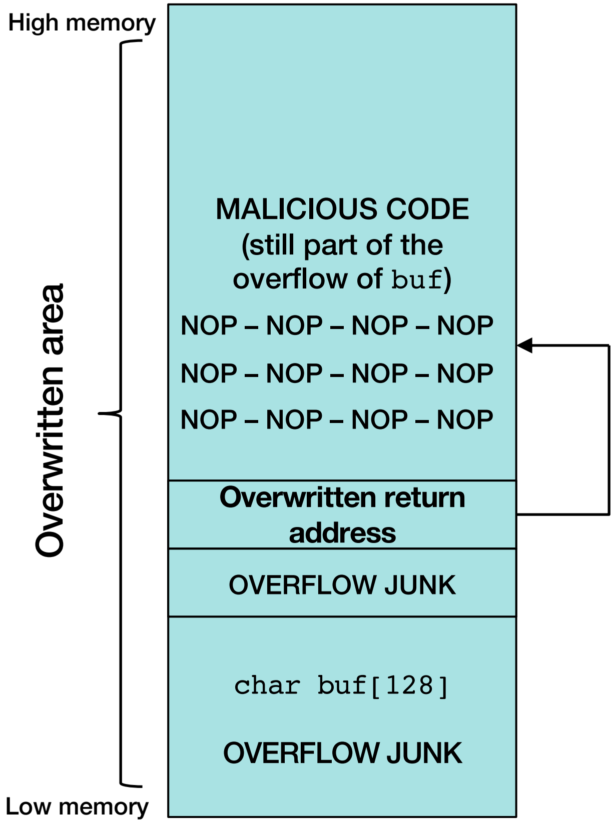


Address Uncertainty

What if we're not sure what the exact address of our injected code is?

NOP slide = NOP sled = landing zone

- Pre-pad the code with a lots of NOP instructions
  - NOP
  - moving a register to itself
  - adding 0
  - etc.
- Set the return address on the stack to any address within the landing zone



Off-by-one overflows

Safe functions aren't always safe

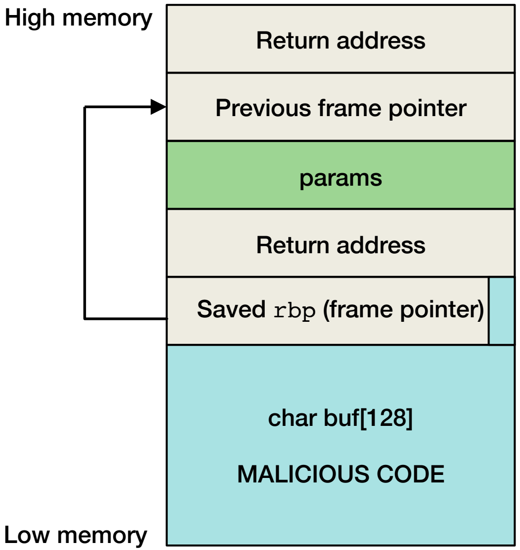
- **Safe counterparts require a count**
  - *strcpy → strncpy*
  - *strcat → strncat*
  - *sprintf → snprintf*
- **But programmers can miscount!**

```
char buf[512];
int i;

for (i=0; i<=512; i++)
    buf[i] = stuff[i];
```

Off-by-one errors

- **We can't overwrite the return address**
- **But we can overwrite one byte of the saved frame pointer**
  - Least significant byte on Intel/ARM systems
    - Little-endian architecture
- **What's the harm of overwriting the frame pointer?**



Off-by-one errors: frame pointer mangling

At the end of a function:

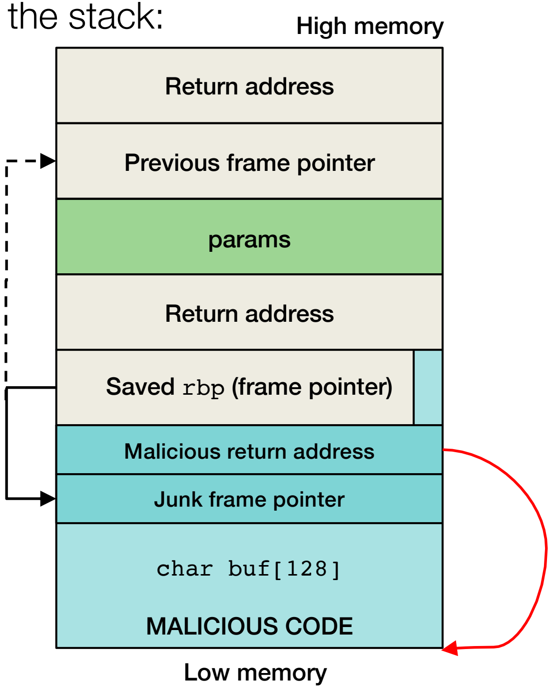
– The compiler resets the stack pointer (%rsp) to the base of the frame (%rbp):
```
mov %rsp, %rbp
```
– and restores the saved frame pointer (which we corrupted) from the top of the stack:
```
pop %rbp     pops corrupted frame pointer into rbp, the frame pointer
ret
```

The program now has the wrong frame pointer when the function returns

The function returns normally –
we could not overwrite the return address

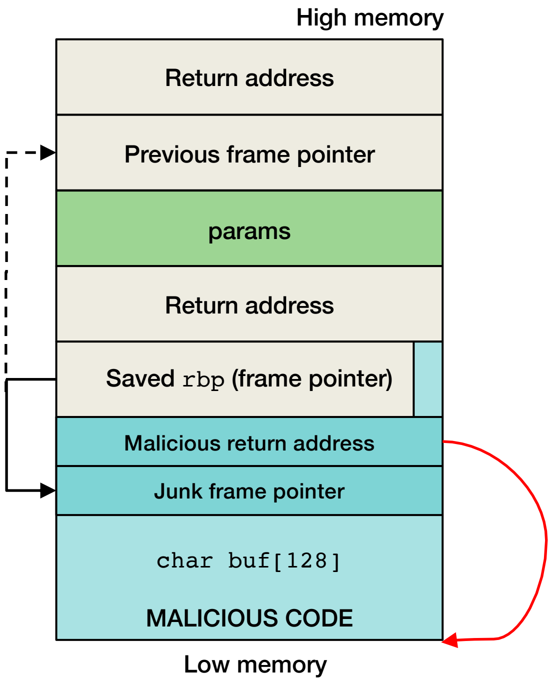
BUT … when the function that called it tries to return, it will update the stack pointer to what it thinks was the valid base pointer and return there:

```
mov %rsp, %rbp     rbp is our corrupted one
pop %rbp           we don't care about the base pointer
ret   return pops the stack from our buffer, so we can jump anywhere
```



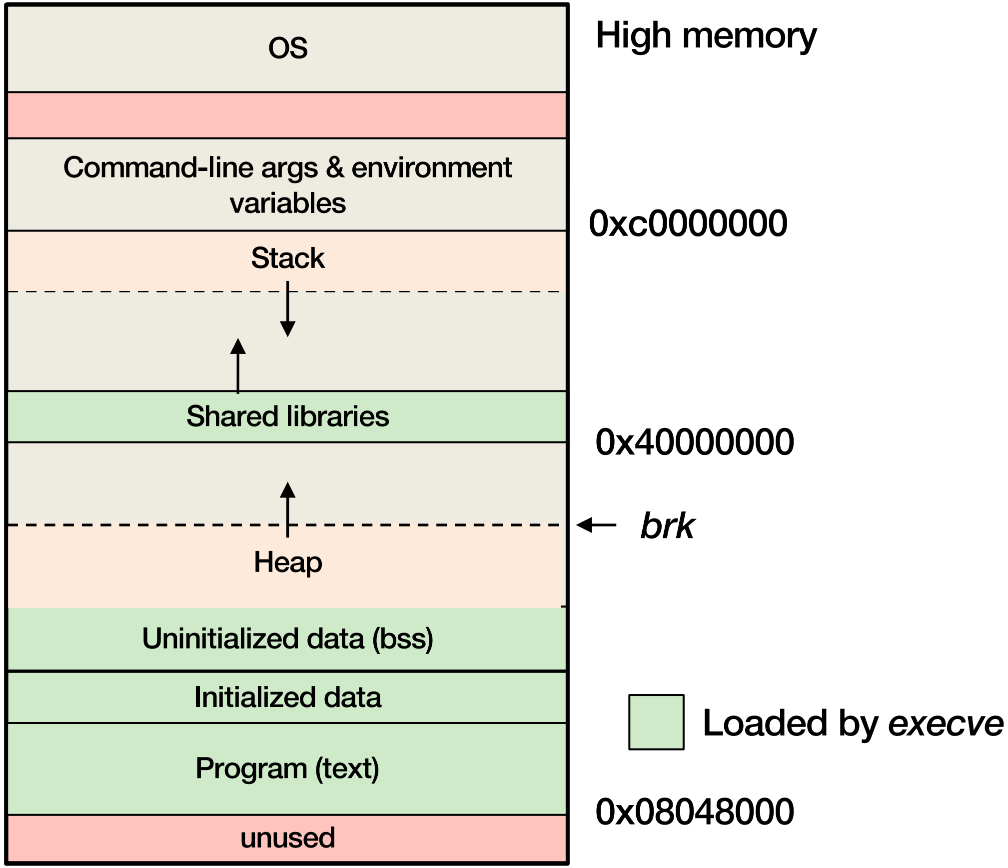
Off-by-one errors: frame pointer mangling

- **Stuff the buffer with**
  - Malicious code, pointed to by ”saved” %rip
  - “saved” %rbp (can be garbage)
  - “saved” %rip (return address)
  - Malicious code, pointed to by ”saved” %rip
- **When the function’s calling function returns**
  - It will return to the “saved” %rip, which points to malicious code in the buffer



Heap & text overflows

Linux process memory map



- Statically allocated variables & dynamically allocated memory (*malloc*) are not on the stack
- Heap data & static data do not contain return addresses
  - No ability to overwrite a return address

Are we safe?

Memory overflow

We may be able to overflow a buffer and overwrite other variables in ***higher*** memory

For example, overwrite a file name

The program

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

char a[15];
char b[15];

int
main(int argc, char **argv)
{
  strcpy(b, "abcdefghijklmnopqrstuvwxyz");
  printf("a=%s\n", a);
  printf("b=%s\n", b);
  exit(0);
}
```

The output
(Linux 4.4.0-59, gcc 5.4.0)

```
a=qrstuvwxyz
b=abcdefghijklmnopqrstuvwxyz
```

Memory overflow – filename example

The program

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

char afile[20];
char mybuf[15];

int main(int argc, char **argv)
{
     strncpy(afile, "/etc/secret.txt", 20);
     printf(”Planning to write to %s\n", afile);
     strcpy(mybuf, "abcdefghijklmnop/home/paul/writehere.txt");
     printf("About to open afile=%s\n", afile);
     exit(0);
}
```

`mybuf` can overflow into `afile`

We overwrote the file name `afile` by writing too much into `mybuf`!

The output (Linux 5.10.63, gcc 8.3.0)

```
Planning to write to /etc/secret.txt
About to open afile=/home/paul/writehere.txt
```

Overwriting variables: changing control flow

- **Even if a buffer overflow does not touch the stack, it can modify global or static variables**

- **Example:**
  - Overwrite a function pointer
  - Function pointers are often used in callbacks

```
int callback(const char* msg)
{
     printf("callback called: %s\n", msg);
}
int main(int argc, char **argv)
{
     static int (*fp)(const char *msg);
     static char buffer[16];

     fp = (int(*)(const char *msg))callback;
     strcpy(buffer, argv[1]);
     (int)(*fp)(argv[2]);     // call the callback
}
```

The exploit

- **The program takes the first two arguments from the command line**
- **It copies `argv[1]` into a buffer with no bounds checking**
- **It then calls the callback, passing it the message from the 2nd argument**

```
int callback(const char* msg)
{
     printf("callback called: %s\n", msg);
}
int main(int argc, char **argv)
{
     static int (*fp)(const char *msg);
     static char buffer[16];

     fp = (int(*)(const char *msg))callback;
     strcpy(buffer, argv[1]);
     (int)(*fp)(argv[2]);     // call the callback
}
```

The exploit

- Overflow the buffer
- The overflow bytes will contain the address you really want to call
  - They're strings, so bytes with 0 in them will not work … making this a more difficult attack

printf attacks

printf and its variants

Standard C library functions for formatted output

- printf: print to the standard output
- wprintf: wide character version of printf
- fprintf, wfprintf: print formatted data to a FILE stream
- sprintf, swprintf: print formatted data to a memory location
- vprintf, vwprintf,vfprintf, vwfprintf :
  print formatted data containing a pointer to argument list

Usage

```
printf(format_string, arguments ...)
printf("The number %d in decimal is %x in hexadecimal\n", n, n);
printf("my name is %s\n", name);
```

Bad usage of printf

Programs often make mistakes with printf

Valid:

```
printf(“hello, world!\n”)
```

Also accepted … but not right

```
char *message = “hello, world\n”);
printf(message);
```

This works but exposes the chance that *message* will be changed

This should be a format string

Dumping memory with printf

```
$ ./tt hello
hello

$ ./tt "hey: %012lx"
hey: 7fffe14a287f
```

printf does not know how many arguments it has.
It deduces that from the format string.

If you don't give it enough, it keeps reading from the stack

We can dump arbitrary memory by walking up the stack

```
$ ./tt %08x.%08x.%08x.%08x.%08x
6d10c308.6d10c320.85d636f0.a1b80d80.a1b80d80
```

```
#include <stdio.h>
#include <string.h>

int
show(char *buf)
{
    printf(buf); putchar('\n');
    return 0;
}

int
main(int argc, char **argv)
{
    if (argc == 2) {
        show(argv[1]);
    }
}
```

Getting into trouble with printf

Have you ever used `%n` ?

Format specifier that will store into memory the number of bytes written so far

```
int printbytes;
printf("paul%n says hi\n", &printbytes);
```

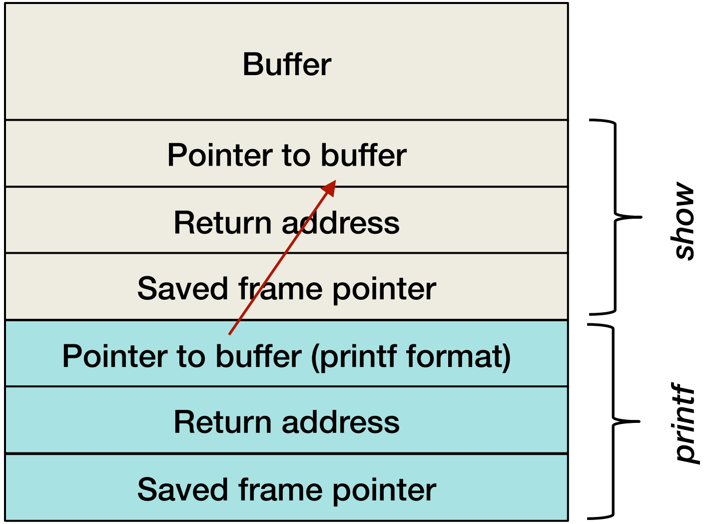
Will print

```
paul says hi
```

and will store the number `4` (which is the value of `strlen("paul")`) into the variable `printbytes`

If we combine this with the ability to change the format specifier, we can write to other memory locations

Bad usage of printf: %n



printf treats this as the 1st parameter after the format string.

- We can skip ints with formatting strings such as %x
- The buffer can contain the address that we want to overwrite

```
#include <stdio.h>
#include <string.h>

int
show(char *buf)
{
    printf(buf);
    putchar('\n');
    return 0;
}

int
main(int argc, char **argv)
{
    char buf[256];

    if (argc == 2) {
        strncpy(buf, argv[1], 255);
        show(buf);
    }
}
```

printf attacks: %n

What good is %n when it's just # of bytes written?

- You can specify an arbitrary number of bytes in the format string

```
printf("%.622404x%.622400x%n" . . .
```

Will write the value 622404+622400 = 1244804 = 0x12fe84

What happens?

- `%.622404x` = write at least 622404 characters for this value
- Each occurrance of `%x` (or `%d`, `%b`, …) will go down the stack by one parameter (usually 8 bytes). We don't care what gets printed
- The `%x` directives enabled us to get to the place on the stack where we want to change a value
- `%n` will write that value, which is the sum of all the bytes that were written

Part 3

Defending against hijacking attacks

Fix bugs

- **Audit software**
- **Check for buffer lengths whenever adding to a buffer**
- **Search for unsafe functions**
  - Use *nm* and *grep* to look for function names
- **Use automated tools**
  - Clockwork, CodeSonar, Coverity, Parasoft, PolySpace, Checkmarx, PREfix, PVS-Studio, PCPCheck, Visual Studio
- **Most compilers and/or linkers now warn against bad usage**

```
tt.c:7:2: warning: format not a string literal and no format arguments [-Wformat-security]
zz.c:(.text+0x65): warning: the 'gets' function is dangerous and should not be used.
```

Fix bugs: Fuzzing

- **Generate semi-random data as input to detect bugs**
  - Locating input validation & buffer overflow problems
  - Enter unexpected input
  - See if the program crashes
- **Enter long strings with searchable patterns**
- **If the app crashes**
  - Search the core dump for the fuzz pattern to find where it died
- **Automated fuzzer tools help with this**
  - E.g., libFuzzer and AFL in C/C++; cargo-fuzz in Rust
- **Or … try to construct exploits using gdb**

Fuzzing in Go

- **Fuzzing available in Go 1.18 (released Feb 2022)**
- **Goal**
  - Make fuzz testing as easy as benchmarking or unit testing
  - No need for custom tools or separate files
- **Seed corpus: user-specified set of inputs to a fuzz test**
  - Fuzzing engine will mutate these inputs to discover new code coverage

https://go.googlesource.com/proposal/+/master/design/draft-fuzzing.md

Don’t use C or C++

- **Most other languages feature**
  - Run-time bounds checking
  - Parameter count checking
  - Disallow reading from or writing to arbitrary memory locations
- **Hard to avoid in many cases**

Specify & test code

- **If it's in the specs, it is more likely to be coded & tested**
- **Document acceptance criteria**
  - "File names longer than 1024 bytes must be rejected"
  - "User names longer than 32 bytes must be rejected"
- **Use safe functions that check allow you to specify buffer limits**
- **Ensure consistent checks to the criteria across entire source**
  - Example, you might `#define` limits in a header file but some files might use a mismatched number.
- **Don't allow user-generated format strings and check results from *printf***

Safer libraries

- **Compilers warn against unsafe *strcpy* or *printf***
- **Ideally, fix your code!**
- **Sometimes you can't recompile (e.g., you lost the source)**
- `libsafe`
  - Dynamically loaded library
  - Intercepts calls to unsafe functions
  - Validates that there is sufficient space in the current stack frame

```
(framepointer – destination) > strlen(src)
```

Dealing with buffer overflows: **No Execute (NX)**

Data Execution Prevention (DEP)

- Disallow code execution in data areas – on the stack or heap
- Set MMU per-page execute permissions to no-execute
- Intel and AMD added this support in 2004

- Examples
  - Microsoft DEP (Data Execution Prevention) (since Windows XP SP2)
  - Linux PaX patches
  - OS X ≥10.5

No Execute – not a complete solution

No Execute Doesn't solve all problems

- Some applications need an executable stack (LISP interpreters)
- Some applications need an executable heap
  - code loading/patching
  - JIT compilers
- Does not protect against heap & function pointer overflows
- Does not protect against `printf` problems

Return-to-libc

- **Allows bypassing need for non-executable memory**
  - With DEP, we can still corrupt the stack … just not execute code from it
- **No need for injected code**
- **Instead, reuse functionality within the exploited app**
- **Use a buffer overflow attack to create a fake frame on the stack**
  - Transfer program execution to the start of a library function
  - `libc` = standard C library … every program uses it!
  - Most common function to exploit: `system`
    - Runs the shell
    - New frame in the buffer contains a pointer to the command to run (which is also in the buffer)
      - E.g., `system("/bin/sh")`

Return Oriented Programming (ROP)

- **Overwrite return address with address of a library function**
  - Does not have to be the start of the library routine
    - Use "borrowed chunks" from a various libraries
  - When the library gets to RET, that location is on the stack, under the attacker's control
- **Chain together sequences ending in RET**
  - Build together "gadgets" for arbitrary computation
  - Buffer overflow contains a sequence of addresses that direct each successive RET instruction
- **It is possible for an attacker to use ROP to execute arbitrary algorithms without injecting new code into an application**
  - Removing dangerous functions, such as system, is ineffective
  - Make attacking easier: use a compiler that combines gadgets!
    - Example: **ROPC** – a Turing complete compiler, `https://github.com/pakt/ropc`

Dealing with buffer overflows & ROP: ASLR

- **Addresses of everything were well known**
  - Dynamically-loaded libraries used to be loaded in the same place each time, as was the stack & memory-mapped files
  - Well-known locations make them branch targets in a buffer overflow attack
- **Address Space Layout Randomization (ASLR)**
  - Position stack and memory-mapped files to random locations
  - Position libraries at random locations
    - Libraries must be compiled to produce position independent code
  - Implemented in
    - OpenBSD, Windows ≥Vista, Windows Server ≥2008, Linux ≥2.6.15, macOS, Android ≥4.1, iOS ≥4.3
  - But … not all libraries (modules) can use ASLR
    - And it makes debugging difficult

Address Space Layout Randomization

- **Entropy**
  - How random is the placement of memory regions?
- **Examples**
  - Linux Exec Shield patch
    - 19 bits of stack entropy, 16-byte alignment > 500K positions
    - Kernel ASLR added in 3.14 (2014)
  - Windows 7
    - 8 bits of randomness for DLLs
      - Aligned to 64K page in a 16MB region: 256 choices
  - Windows 8
    - 24 bits for randomness on 64-bit processors: >16M choices

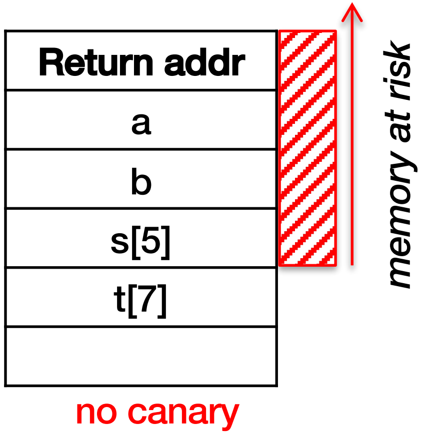
Dealing with buffer overflows: Canaries

Stack canaries

– Place a random integer before the return address on the stack
– Before a return, check that the integer is there and not overwritten: a buffer overflow attack will likely overwrite it

```
int a, b=999;
char s[5], t[7];

gets(s);
```



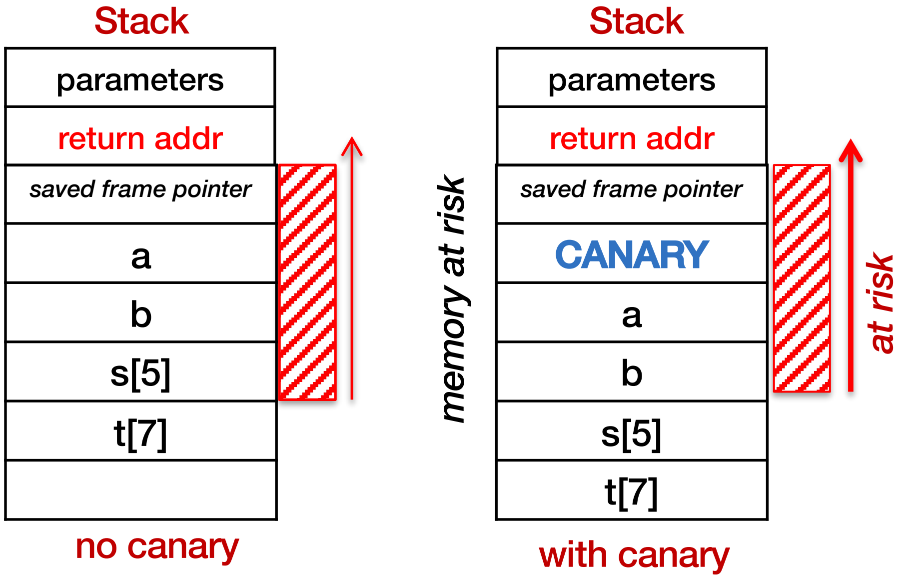
Dealing with buffer overflows: Canaries

Stack canaries

- Place a random integer before the return address on the stack
- Before a return, check that the integer is there and not overwritten: a buffer overflow attack will likely overwrite it

```
int a, b=999;
char s[5], t[7];

gets(s);
```



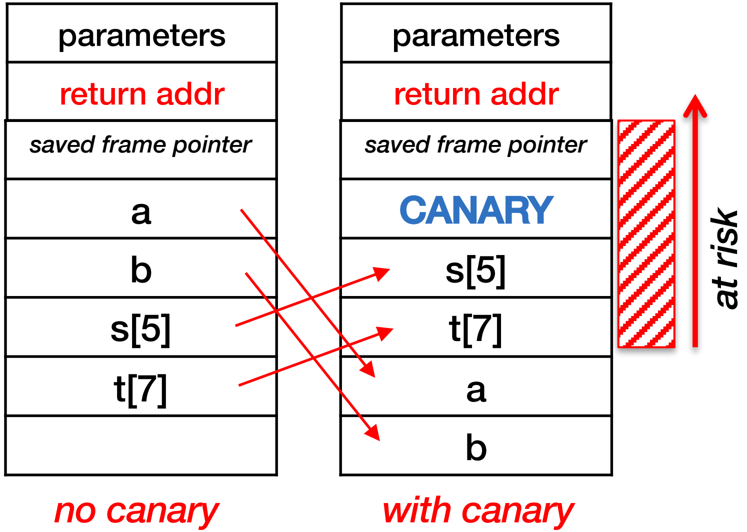
Refining Stack Canaries: ProPolice

IBM's ProPolice gcc patches

- Allocate arrays into higher memory in the stack
- Ensures that a buffer overflow attack will not clobber non-array variables
- Increases likelihood that the overflow won't attack the logic of the current function

```
int a, b=999;
char s[5], t[7];

gets(s);
```



Stack canaries

- **Again, not foolproof**
- **Heap-based attacks are still possible**
- **Performance impact**
  - Need to generate a canary on entry to a function and check canary prior to a return
  - Minimal degradation ~8% for apache web server

Intel CET: Control-Flow Enforcement Technology

Developed by Intel & Microsoft to thwart ROP attacks

- Available with the Tiger Lake microarchitecture (mid-2020)

Two mechanisms

1. **Shadow stack**
   - Secondary stack
     - Only stores return addresses
     - MMU attribute disallows use of regular *store* instructions to modify it
   - Stack data overflows cannot touch the shadow stack – cannot change control flow

2. **Indirect branch tracking**

Intel CET: Control-Flow Enforcement Technology

Indirect Branch Tracking

- Restrict a program's ability to use jump tables

- Jump table = table of memory locations the program can branch
  - Used for switch statements & various forms of lookup tables

- Jump-Oriented Programming (JOP) and Call Oriented Programming (COP)
  - Techniques where attackers abuse JMP or CALL instructions
  - Like Return-Oriented Programming but use gadgets that end with indirect branches

- New `ENDBRANCH` (ENDBR64) instruction allows a programmer to specify valid targets for indirect jumps
  - If you take an indirect jump, it has to go to an ENDBRANCH instruction
  - If the jump goes anywhere else, it will be treated as an invalid branch and generate a fault

Heap attacks – pointer protection

- **Encrypt pointers (especially function pointers)**
  - Example: XOR with a stored random value
  - Any attempt to modify them will result in invalid addresses
  - XOR with the same stored value to restore original value
- **Degrades performance when function pointers are used**

Hardware Attacks: Example - Rowhammer

DDR4 memory protections are broken wide open by new Rowhammer technique

ars technica

Researchers build "fuzzer" that supercharges potentially serious bitflipping exploits.

Dan Goodin • 11/15/2021

Rowhammer exploits that allow unprivileged attackers to change or corrupt data stored in vulnerable memory chips are now possible on virtually all DDR4 modules due to a new approach that neuters defenses chip manufacturers added to make their wares more resistant to such attacks.

Rowhammer attacks work by accessing—or hammering—physical rows inside vulnerable chips millions of times per second in ways that cause bits in neighboring rows to flip, meaning 1s turn to 0s and vice versa. Researchers have shown the attacks can be used to give untrusted applications nearly unfettered system privileges, bypass security sandboxes designed to keep malicious code from accessing sensitive operating system resources, and root or infect Android devices, among other things.

https://arstechnica.com/gadgets/2021/11/ddr4-memory-is-even-more-susceptible-to-rowhammer-attacks-than-anyone-thought/

Hardware Attacks: Example - Rowhammer

- **New attack technique discovered**
  - Uses non-uniform patterns that access two or more rows with different frequencies
  - Bypasses all defenses built into memory hardware
  - 80% of devices can be hacked this way
  - Cannot be patched!
- **Sample attacks**
  - Gain unrestricted access to all physical memory by changing bits in the page table entry
  - Give untrusted applications root privileges
  - Extract encryption key from memory

The end

The End

Top Software Weaknesses for 2020

MITRE, a non-profit organization that manages federally-funded research & development centers, publishes a list of top security weaknesses

Rank	Name	Score
1	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.81
2	Out-of-bounds Write	46.17
3	Improper Input Validation	33.47
4	Out-of-bounds Read	26.50
5	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
6	SQL Injection	20.69
7	Exposure of Sensitive Information to an Unauthorized Actor	19.16
8	Use After Free	18.87
9	Cross-Site Request Forgery (CSRF)	17.29
10	OS Command injection	16.44

https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html