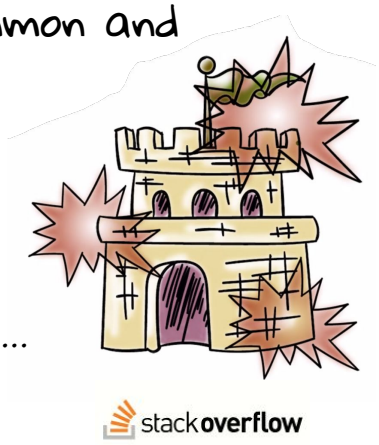
Software Security
Lesson Introduction

- Software vulnerabilities and how attackers exploit them.
- Defenses against attacks that try to exploit buffer overflows.
- Secure programming: Code "defensively", expecting it to be exploited. Do not trust the "inputs" that come from users of the software system.

Software Vulnerabilities & How They Get Exploited

- Example: Buffer overflow - a common and persistent vulnerability



- Stack buffer overflows
- Stacks are used...
  - in function/procedure calls
  - for allocation of memory for...
    - -local variables
    - -parameters
    - -control information (return address)

- A **stack overflow** is a type of **buffer overflow** error that
- occurs when a computer program tries to use more memory space than has been allocated to that stack.

https://www.fortinet.com/resources/cyberglossary/buffer-overflow

- **buffer overflow (AKA buffer overrun)** occurs when the amount of data in the buffer exceeds its storage capacity.
- That extra data **overflows** into adjacent memory locations and corrupts or overwrites the data in those locations.

https://en.wikipedia.org/wiki/Stack_Overflow

- The StackOverflow website was created by Jeff Atwood and Joel Spolsky in 2008.
- The name for the website was chosen by voting in April 2008 by readers of Coding Horror, Atwood's programming **blog**.
- On 31 July 2008, Jeff Atwood sent out invitations encouraging his **subscribers** to take part in the private beta of the new website, limiting its use to those willing to test out the new software.
- On 15 September 2008 it was announced that the public beta version was in session and that the general public was now able to use it to seek assistance on programming related issues.
- The design of the Stack Overflow logo was decided by a voting process

A Vulnerable Password Checking Program

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

int main(int argc, char *argv[]) {
        int allow_login = 0;
        char pwdstr[12];
        char targetpwd[12] = "MyPwd123";
        gets(pwdstr);
        if (strncmp(pwdstr,targetpwd, 12) == 0)
                allow_login = 1;

        if (allow_login == 0)
                printf("Login request rejected");
        else
                printf("Login request allowed");
}
```

Attacker Code Execution

We type a correct password (MyPwd123) of less than 12 characters:

The login request is allowed.

Now let us type "BadPassWd" when we are asked to provide the password:

The login request is rejected.

Attacker Bad Input Quiz

How can we attack this code and login without knowing the password?

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

int main(int argc, char *argv[]) {
        int allow_login = 0;
        char pwdstr[12];
        char targetpwd[12] = "MyPwd123";
        gets(pwdstr);
        if (strncmp(pwdstr,targetpwd, 12) == 0)
                allow_login = 1;

        if (allow_login == 0)
                printf("Login request rejected");
        else
                printf("Login request allowed");
}
```

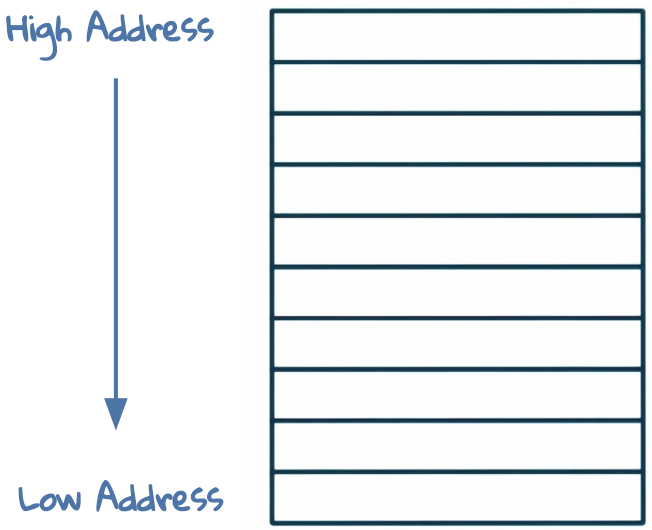
Stack Access Quiz

Check the lines of code, when executed, accesses addresses in the stack frame for main():

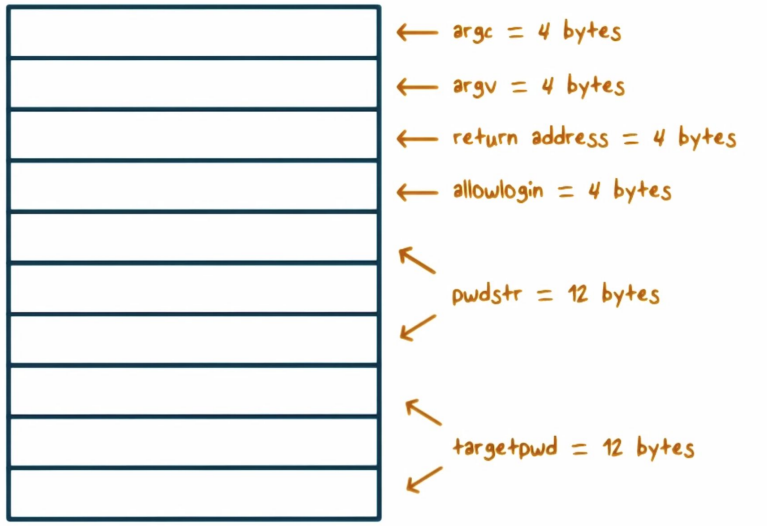
```
int main(int argc, char *argv[]) {
    int allow_login = 0;
    char pwdstr[12];
    char targetpwd[12] = "MyPwd123";
    gets(pwdstr);
    if (strncmp(pwdstr,targetpwd, 12) == 0)
        allow_login = 1;

    if (allow_login == 0)
        printf("Login request rejected");
    else
        printf("Login request allowed");
}
```

Understanding the Stack

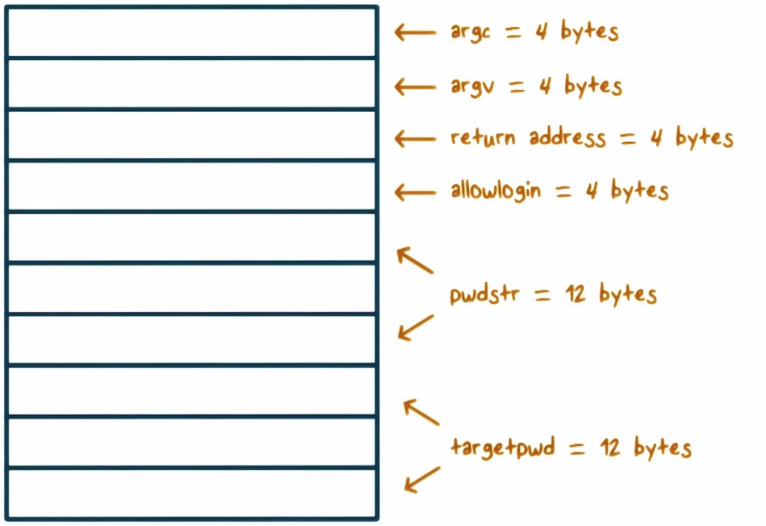


Attacker Code Execution



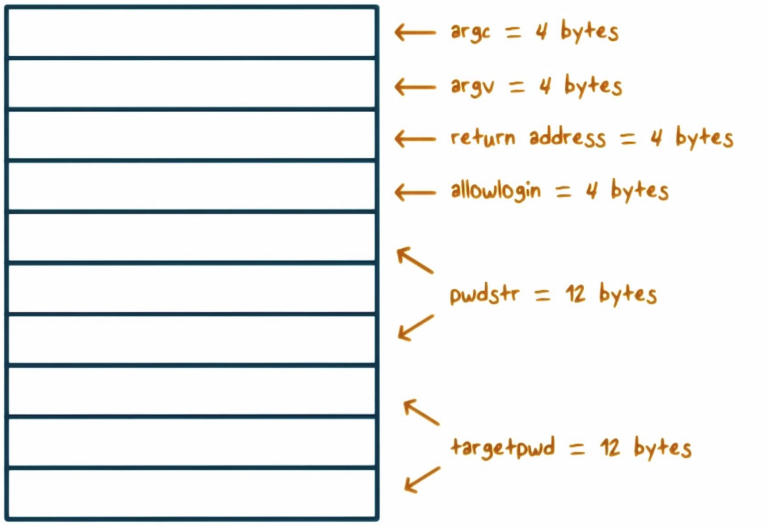
Attacker Code Execution

If we type a really long string, we will overflow into the return address space.



Attacker Code Execution

We can carefully overflow the return address so it contains the value of an address where we put some code we want executed.



ShellCode

Shell Code: creates a shell which allows it to execute any code the attacker wants.

Whose **privileges** are used when attacker code is executed?

- The host program's
- System service or OS root privileges

LEAST Privilege is IMPORTANT

Return-to-libc

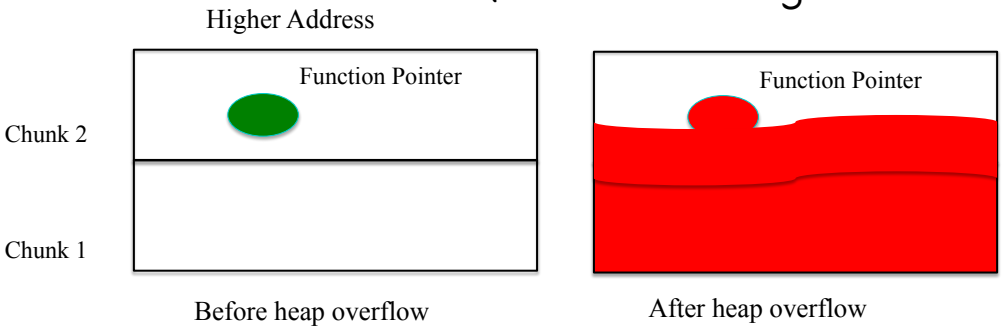
- **Return-to-libc**: the return address is overwritten to point to a standard library function.

Variations of Buffer Overflow (1)

- Heap
  - does not have a return address
  - So you cannot hijack the control flow of the program
- Heap Overflows:
  - Data stored in the heap is overwritten.
  - Data can be tables of function pointers.

•Heap Overflow - Example

- Buffer overflows that occur in the heap data area.
- Overwrite the function pointer in the adjacent buffer



Before heap overflow

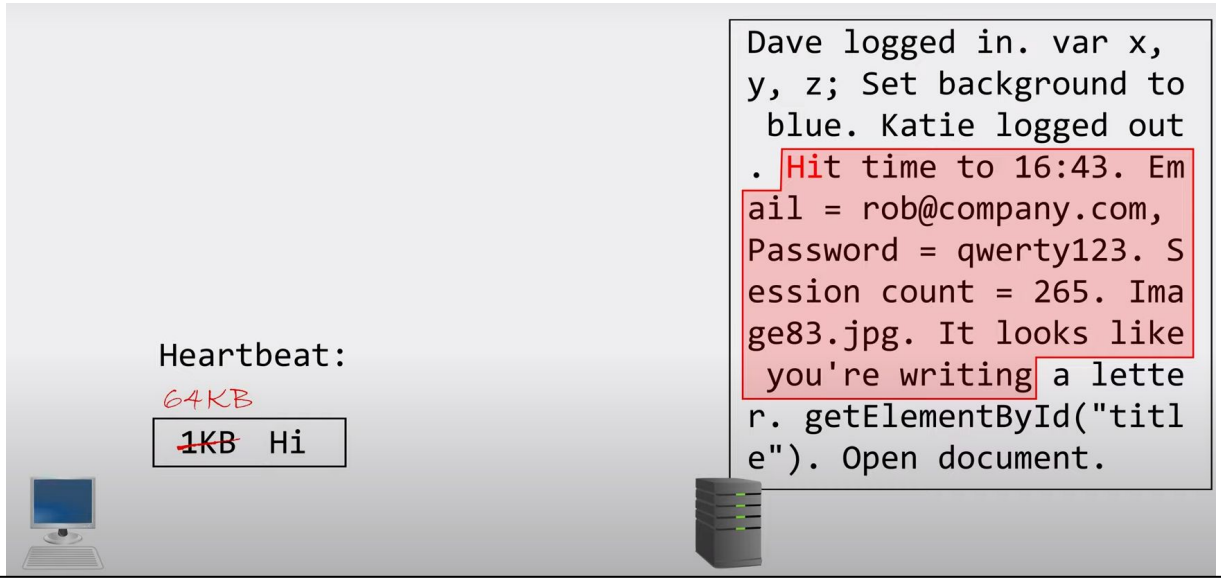
After heap overflow

Variations of Buffer Overflow (2)

- OpenSSL Heartbleed Vulnerability:
  - classified as a buffer over-read
  - TLS implementations other than OpenSSL, such as
    - GnuTLS and
    - Windows implementation,
    - were not affected
  - read much more of the buffer than just the data, which may include sensitive data.

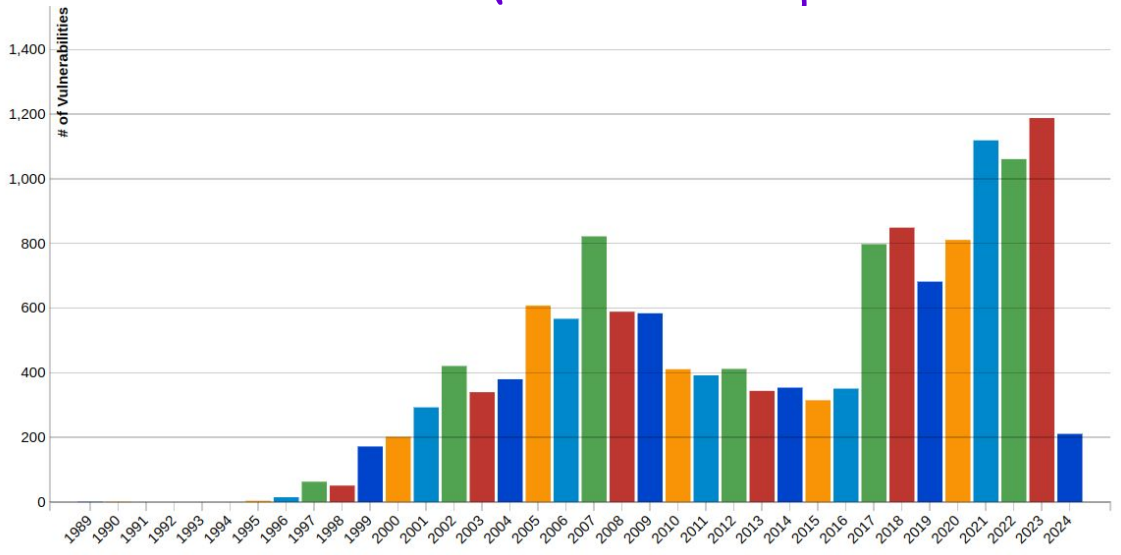
OpenSSL Heartbleed Vulnerability:

From Heartbeat to Heartbleed

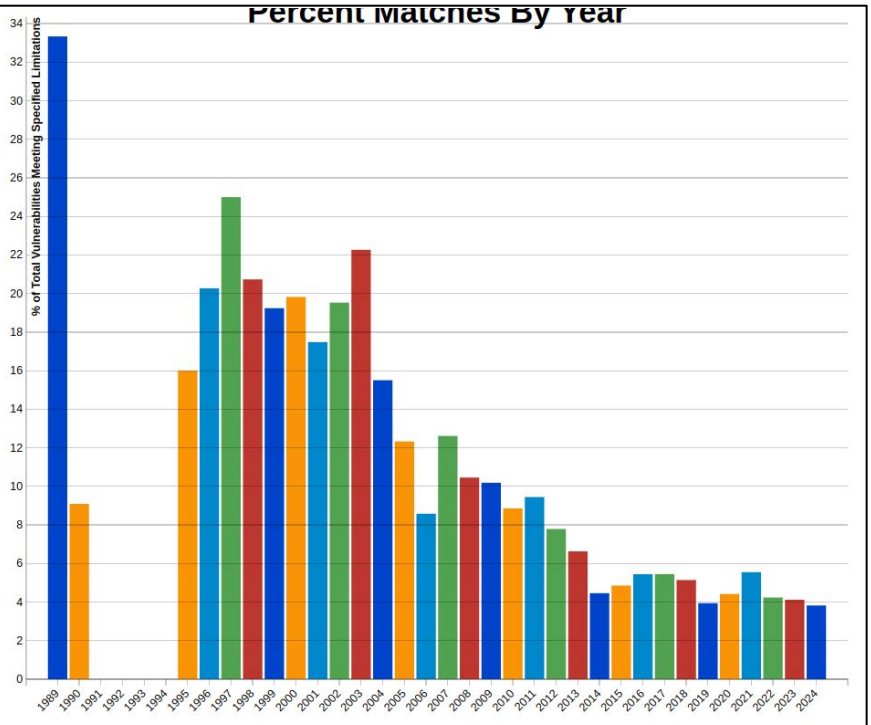


National Vulnerability Database (NVD)

Buffer overflow statistics https://web.nvd.nist.gov/view/vuln/search



According to NVD:
Buffer overflow
statistics (Percentage



Year	Matches	Total	Percentage
1989	1	3	33.33%
1990	1	11	9.09%
1991	0	15	0.00%
1992	0	13	0.00%
1993	0	13	0.00%
1994	0	25	0.00%
1995	4	25	16.00%
1996	15	74	20.27%
1997	63	252	25.00%
1998	51	246	20.73%
1999	172	894	19.24%
2000	202	1019	19.82%
2001	293	1676	17.48%
2002	421	2156	19.53%
2003	340	1527	22.27%
2004	380	2451	15.50%
2005	608	4932	12.33%
2006	567	6608	8.58%
2007	822	6516	12.62%
2008	589	5632	10.46%
2009	584	5732	10.19%
2010	411	4639	8.86%
2011	392	4150	9.45%
2012	412	5288	7.79%
2013	344	5187	6.63%
2014	354	7937	4.46%
2015	315	6487	4.86%
2016	351	6447	5.44%
2017	798	14643	5.45%
2018	849	16509	5.14%
2019	682	17305	3.94%
2020	811	18350	4.42%
2021	1119	20155	5.55%
2022	1061	25050	4.24%
2023	1188	28829	4.12%
2024	211	5587	3.78%

About 15K Buffer overflow in NVD from a total of 230K vulnerabilities

Defense Against Buffer Overflow Attacks

Programming language choice is crucial.

Examples of safe languages:
Java, C#

https://www.fortinet.com/resources/cyberglossary/buffer-overflow

- Nearly all apps, web servers, and web app environments are vulnerable to buffer overflows.
- Environments that are written in interpreted languages, such as Java and Python, are immune to the attacks, with the exception of overflows in their interpreter.

Defense Against Buffer Overflow Attacks

Why are some languages safe?

- Buffer overflow becomes nearly impossible due to runtime system checks

The drawback of secure languages

- Possible performance degradation

Strongly vs. Weakly Typed Language Quiz

Strongly typed languages help reduce software vulnerabilities. Determine which of the following options apply to strongly typed languages and which are for weakly typed. (Use 's' or 'w').

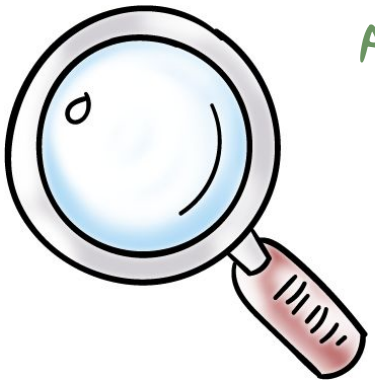
- ☐ Any attempt to pass data of incompatible type is caught at compile time or generates an error at runtime.
- ☐ An array index operation b[k] may be allowed even though k is outside the range of the array.
- ☐ It is impossible to do "pointer arithmetic" to access arbitrary area of memory.

Defense Against Buffer Overflow Attacks

When Using Unsafe Languages:

- Check input (ALL input is EVIL)
- Use automatic tools to analyze code for potential unsafe functions.

Defense Against Buffer Overflow Attacks



Analysis Tools...

- Can flag potentially unsafe functions
- Can help mitigate security lapses, but it is really hard to eliminate all buffer overflows.

Examples of analysis tools can be found at:
https://www.owasp.org/index.php/Source_Code_Analysis_Tools

Thwarting Buffer Overflow Attacks

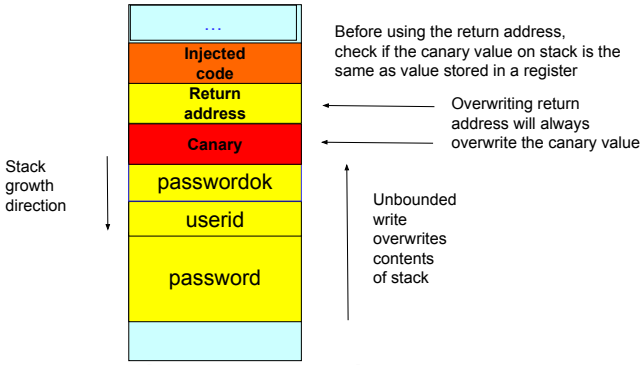
Stack Canaries:

- When a return address is stored in a stack frame, a random canary value is written just before it.
- Any attempt to rewrite the address using buffer overflow will result in the canary being rewritten and an overflow will be detected.

- A copy of stack canary is saved in a randomized and guarded memory.
  - Whenever a unsafe stack object is used, a canary check function will be called before return instruction.
- The canary value is randomly generated for each function call.
  - Therefore, the attacker cannot know the actual canary value.
  - The attack code cannot "read" this value and then use for injection because the attack code need to be injected before it can be executed to do anything (including reading the canary value).

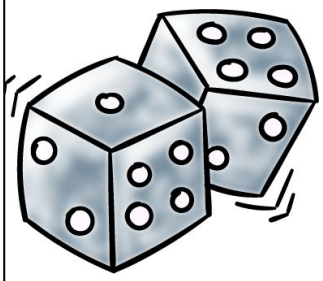
•Countermeasure - Stack Protection

- Canary for tamper detection



- No code execution on stack

Thwarting Buffer Overflow Attacks



- Address Space Layout Randomization (ASLR)
  - randomizes stack, heap, libc, etc.
  - This makes it harder for the attacker to find important locations
    - (e.g., libc function address).
- Use non-executable stack coupled with ASLR