# Source Code Auditing: Day 1

## Penetration Testing & Vulnerability Analysis

Brandon Edwards

brandon@isis.poly.edu

# Agenda

- Introduction
- Goals
- What is Source Code Auditing
- Methodology
- Bug Classes
- Tools

# Introduction

- I read lots of code, I like finding bugs
- Bugs can be grown into exploits
- Exploits let you own stuff
- Developer souls go in the jar

# Day 1

- Scoping, targeting a large code base
- Using navigation tools / auditing tools
- See what vulnerabilities look like in code
- Cover some basic bug classes
- Mostly C/C++

# Source Code Auditing

- The process of reading source code to find vulnerabilities
- Great for finding implementation bugs
  - Memory Corruption
  - In-line Injection (SQL, filenames)
  - Race Conditions
- Compliments a design review

# Source Code Auditing

- Tedious
- Hard to estimate time required
- Thorough understanding of the language is absolutely necessary

# Methodology

Finding and attacking valuable targets

# Methodology Approaches

- There can be different reasons to audit code…
- To find the most bugs possible?
- To find the easiest, most reliable bug to exploit?

- For the sake of offense, we are looking for the latter

# Methodology

- There is NEVER enough time
  - Large code bases
  - Timed engagement
  - Lifetime of target data value

- To rephrase that, you can never have your **0**day too soon ;)
- Since time is critical, so is having an attack plan
- I use methodology to mean attack plan for approaching the app

# Methodology

- Understand the application
  - Review documentation, understand purpose
  - Examine the attack surface
  - From the attack surface, identify target components

- If you've already done a design + operational review, the above is much easier

# Targeting Components

- Traditional
  - Input sources to related code paths
  - Security mechanisms
  - Complex parsing, protocols, data management

- Meta Targeting
  - Comments indicating complexity or difficulty
  - FIXME
  - XXX
  - Swearing
  - Typos

# Meta Targeting

- Old Code (Copyright 1998, etc)
- Code checked in at inopportune times
  - 2 AM near date of shipping
  - At the same time that other buggy code was checked in
- Patterns from other buggy code
- Profile developers for bad code, watch where else they check in

# Grep Targeting

- Looking for bad APIs, then checking if they're vulnerable
- Can find lots of "potential" bugs
- May find bugs in useless components ☹
- Can be painful to trace back to input
- More useful if performed within a confined context

# Grep In General

- Is great for finding keywords…
- These keywords can help you find things of value to review
  - Bad APIs
  - Code related to keyword concepts
- Grep alone will not give you an understanding of the code
- Without understanding, most awesome bugs are missed
- It is not a golden unicorn

# Reading Code

- Initial reading can be frustrating
- Understanding components requires context
- Read code iteratively to get a grasp on context
- The trick is learning to skim unimportant code
- You can always return to a previously skipped piece of code if it becomes relevant

# Reading Code Quickly

- Game of mental pattern matching; some intuition
- How to see the important?
- You'll start to get an "eye" for things that actually do important things, like
  - Copy or move data around
  - Perform Input/Output
  - Funny smelling things

# Reading Code Quickly

- How to know when to skim
- Getting an "eye" for things which are not important, ignoring until they gain importance
- There is lots of filler code which will not relate to the code you are interested in

# Reading Code Quickly

- For example, most of these you can skip until they relate to a target component
  - Function prototypes
  - #define macros
  - Hard coded value assignment
  - Initial value assignment
  - Abstraction

- This does not mean these are all unimportant
- These just probably require less detailed attention until a deeper understanding is gained of how the code uses them

# Reading Code Quickly

- Take the initial value assignment as an example..
- Failure to assign initial values can lead to vulnerabilities

```
int doAuth(char *name, char *pw)
{
    int auth;

    if (userIsBanned(name))
    {
        auth = 0;
    }

    if (authenticateUser(name, pw))
    {
        auth = 1;
    }

    return auth;
}
```

# Reading Code Quickly

- In the previous example the lack of initialization became important due to how the variable was used
- If you spot an uninitialized variable being used, jump on it, there is probably a vulnerability!
- That being said, verifying every variable is initialized is a waste of time

# Reading Code Quickly

- Abstraction is another great example – lots of bug potential!
- There will be misuses of it leading to vulnerabilities
- But developers who like abstraction, love abstraction; so it is probably used in every possible place
- There is a lot of abstraction filler code you can skim
- Think more about their patterns in misusing it, look for those patterns

# Code Auditing Tools

- Various tools exist to aid in auditing
- Editors / reading tools
- Pattern matching tools
- Static analyzers
- Pen & pad ;)

# Code Reading Tools

- Free syntax aware tools
  - VI / VIM
  - Emacs
  - Notepad++
  - Source Navigator
  - Eclipse
- Commercial Products
  - Visual Studio
  - Understand
  - Source Insight

# Static Analysis Tools

- Commercial Tools
  - Fortify
  - Klocwork
  - Coverity
- Free Products
  - LLVM Clang Static Analyzer
  - FindBugs (Java)
  - RATS (more like fancy pattern matching)

# Reading Code: Tools

- Static analyzers are cute
  - Often miss vulnerabilities
  - Will have many false positives

- They can aid in understanding a code base..
- They can not replace the mind

# Reading Code: Tools

- Highlighted editors & navigators are useful
- Basic useful functionality in auditing tools:
  - Tracking where variables defined
  - Tracking where functions are implemented
  - Function call-graphing
  - Generic search/regex codebase
- As a free tool, I like Source Navigator

# Implementation Bugs

Where the bling happens

# Implementation Bugs

- A bug in how the code was implemented, which can allow an attacker to cause the application to deviate from its design

- These are commonly caused by:
  - Failure to validate input
  - Misuse or misunderstanding of an API
  - Miscalculation of an operation
  - Failure to verify the result of an operation
  - Application state failures

# Implementation Bugs

- Notoriously affect complex code or parsing
  - Complex file formats & protocols can be difficult to properly implement
  - Trusting, or assuming the structure or validity of input
  - Failure to track relationships, such as object references

# Memory Corruption

# Memory Corruption

"Memory corruption happens when the contents of a memory location are unintentionally modified due to programming errors. When the corrupted memory contents are used later in the computer program, it leads either to program crash or to strange and bizarre program behavior." – Wikipedia

# Memory Corruption

- "program crash" and "…strange program behavior" is fancy developer speak for "busticated and exploitable"
- Has been infamously responsible for vulnerabilities
- "There is no such thing as a crash or DoS, there are only vulnerabilities which ***you*** can not exploit"

  – Ryan Permeh

# Memory Corruption

- The classic implementation bug
- Has been infamously responsible for vulnerabilities
- Can result in the complete compromise of the application, and in turn the machine
- Been public since the 1980's, still happen today

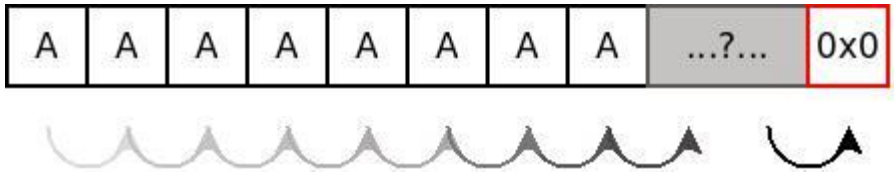
# Memory Corruption

- Basic example..

```
int vuln_function(char *userstring)
{
    char buf[128];
    /* make copy of data to manipulate */
    strcpy(buf, userstring);
    /* ... */
    return;
}
```

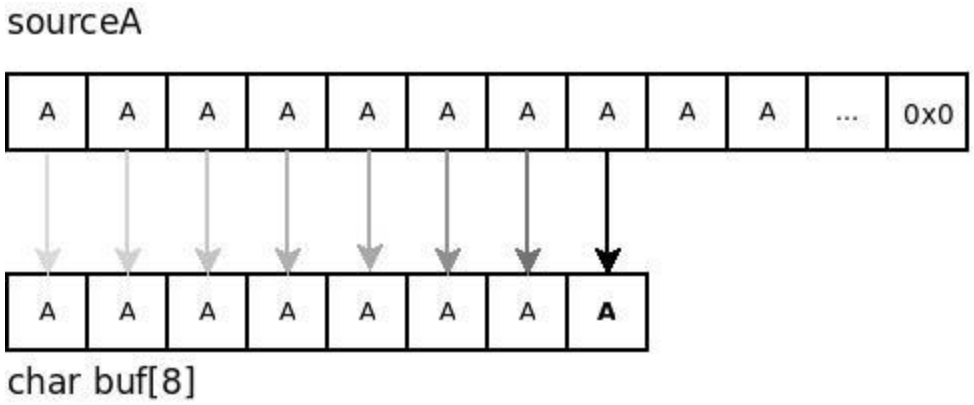
# Memory Corruption

- The strcpy() function performs a copy from a source string to a destination string
- The string is determined by a sequence of bytes terminated by a NULL byte



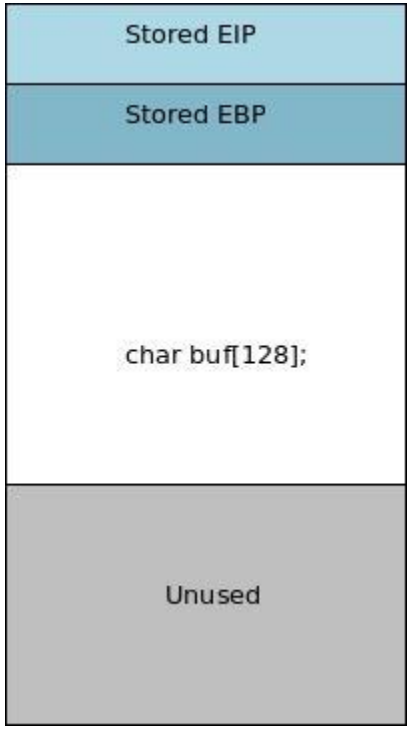
# Memory Corruption

- In this example, it copies from an argument onto the function's local stack

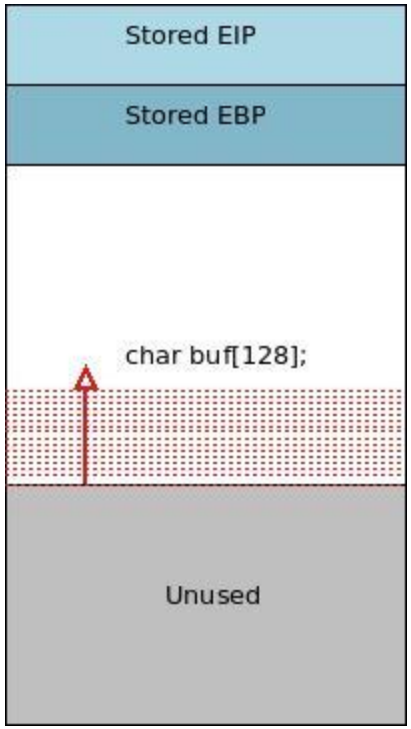


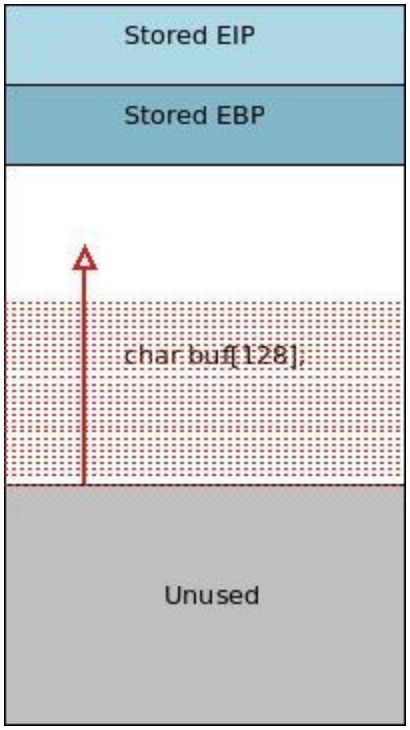

- Because the copy is unbounded, if there is more data in the source than there is space available, it will overwrite data on the stack
- This means if an attacker could reach this, they could overwrite contents on the stack

# Memory Corruption

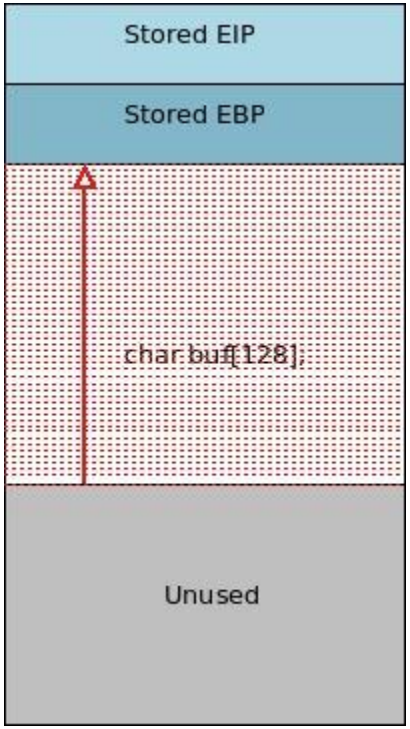


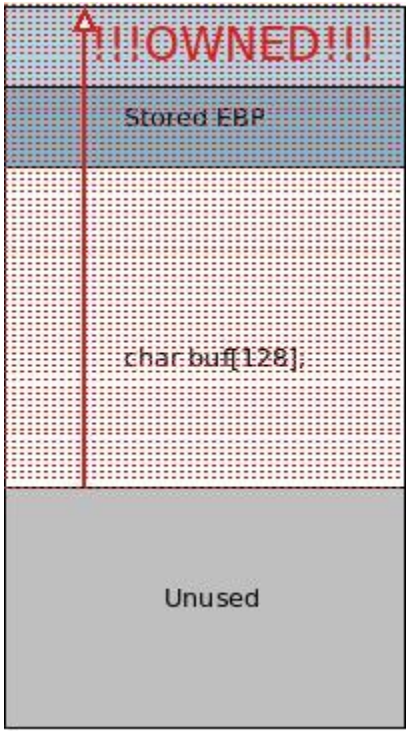
# Memory Corruption



# Memory Corruption



# Memory Corruption

- Trivial example
- While this example shows overwriting the stack, corruption of other memory regions such as heap or BSS are often exploitable
- More is taught on these concepts later in the course by Dino Dai Zovi

# Memory Corruption

- Lots of memory corruption can happen from byte-by-byte copies
- Lots of APIs do this, not just strcpy()
  - strcpy()
  - strcat()
  - sprintf()
  - gets()
  - …list goes on

# Memory Corruption

- Homegrown byte-by-byte copies

```
int vuln_function(char *userstring)
{
    char buf[128];
    char *src, *dst;
    src = userstring;
    dst = buf;
    while(*src != 0x0)
    {
        *dst++ = *src++;
    }
    /* ... */
    return;
}
```

# Memory Corruption

- So.. Unbounded data copying is bad..
- Newer, safer APIs do exist to allow developers to specify the amount of data to be copied
- strcpy() probably unlikely to be seen in modern code, but easy to grep for and find in your audits
- Memory corruption from pointer loops copying data can still be found, so examine any place this is seen

# Memory Corruption

- Just because a better API exists, does not mean it is used properly
- Consider strncpy() instead..
- Bounded string copy:

```
char *strncpy(char *dest, const char *src, size_t n);
```

- Now there's a parameter to limit length of data copied

# Memory Corruption

- The length variable can be completely misunderstood or misused

```
int vuln_function(char *userstring)
{
    char buf[128];
    strncpy(buf, userstring,strlen(userstring));
    /* ... */
    return;
}
```

# Memory Corruption

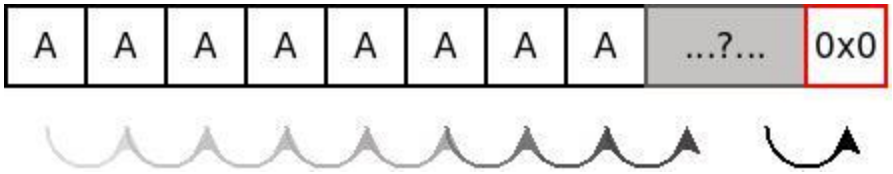
- The length does not account for NULL termination

```
int vuln_function(char *userstring)
{
    char buf[128];
    strncpy(buf, userstring, sizeof(buf));
    /* ... */
    return;
}
```

- If amount of data to copy is greater than or equal to size of buf, no NULL byte will be placed by strncpy()

# Memory Corruption

- Remember, C string functions need there to be a NULL byte to know where a string ends…




- Later on, code may be assuming the string is only as long as the sizeof(buf), when in reality the string is as long as wherever the next NULL is in memory
- This could be an adjacent piece of memory the attacker controls, such as another buffer declared on the stack

# Memory Corruption

- Bugs from copy & pasting

```
int vuln_function(char *string)
{
    char buf1[256];
    char buf2[256];
    char buf3[128];
    /* … */
    strncpy(buf1, string, sizeof(buf1)-1);
    strncpy(buf2, string, sizeof(buf1)-1);
    /* … */
    strncpy(buf3, string, sizeof(buf1)-1);
    return;
}
```

# Memory Corruption

- Lots of developers have learned to consider the size and NULL byte
- Some still confuse or forget which functions do what
- You can find this in string concatenation functions like strcat()

```
char *strncat(char *dest, const char *src, size_t n);
```

- This function appends a string from the source buffer to the destination buffer, adding to the end of an existing C string in dest

# Memory Corruption

- Note that strncat() size parameter n does not account for data already in the destination buffer

```
int vuln_function(char *string)
{
    char buf1[256];
    strncat(buf1, “static data”, sizeof(buf1)-1);
    /* … */
    strncat(buf1, string, sizeof(buf1)-1);
    return;
}
```

- If there is already data in buf1, it can overwrite beyond the buffer!

# Memory Corruption

- Other common misunderstandings of size can happen with wide-characters, like wchar_t

"Under Win32, wchar_t is 16 bits wide and represents a UTF-16 code unit. On Unix-like systems wchar_t is commonly 32 bits wide and represents a UTF-32 code unit."

–Wikipedia

# Memory Corruption

```
size_t mbstowcs(wchar_t *dest, const char *src, size_t n);
```

- Size miscalculation can happen by not considering that sizeof() returns count of 8bit chars, and wchar_t is larger than that:

```
int vuln_function(char *string1)
{
    wchar_t buf1[256];
    mbstowcs(buf1, string1, sizeof(buf1)-1);
    return;
}
```

# Memory Corruption

- The size length is given as sizeof()…
- The problem is, the size argument for mbstowcs() is the **count** of wide characters to write
- Wide characters are bigger than bytes:

```
wchar_t buf1[256];
mbstowcs(buf1, string1, sizeof(buf1)-1);
```

- On Windows, where wchar_t is 16bits, sizeof(buf) is really 512
- Example code results in a copy of 511 wide-characters into the destination buffer, when it was intended to be 255

# Auditing Tips

- Be aware of the types of strings being used by the application, and review how the string functions are used
- Never assume that just because an intent was made to be secure (using safer APIs, mindfulness of memory size) that the code does not have bugs
- Grepping around for string manipulation APIs can be a quick way to find pieces of code may be of interest
- **Always review the arithmetic used for string size calculation and copying**

# Data Types

# Data Type Bugs

- Data types are fundamental to programming
- Data types use a specific binary format to represent a finite amount of data
- Data types are often misunderstood or mishandled
- This misunderstanding can introduce vulnerabilities

For this section x86 is assumed

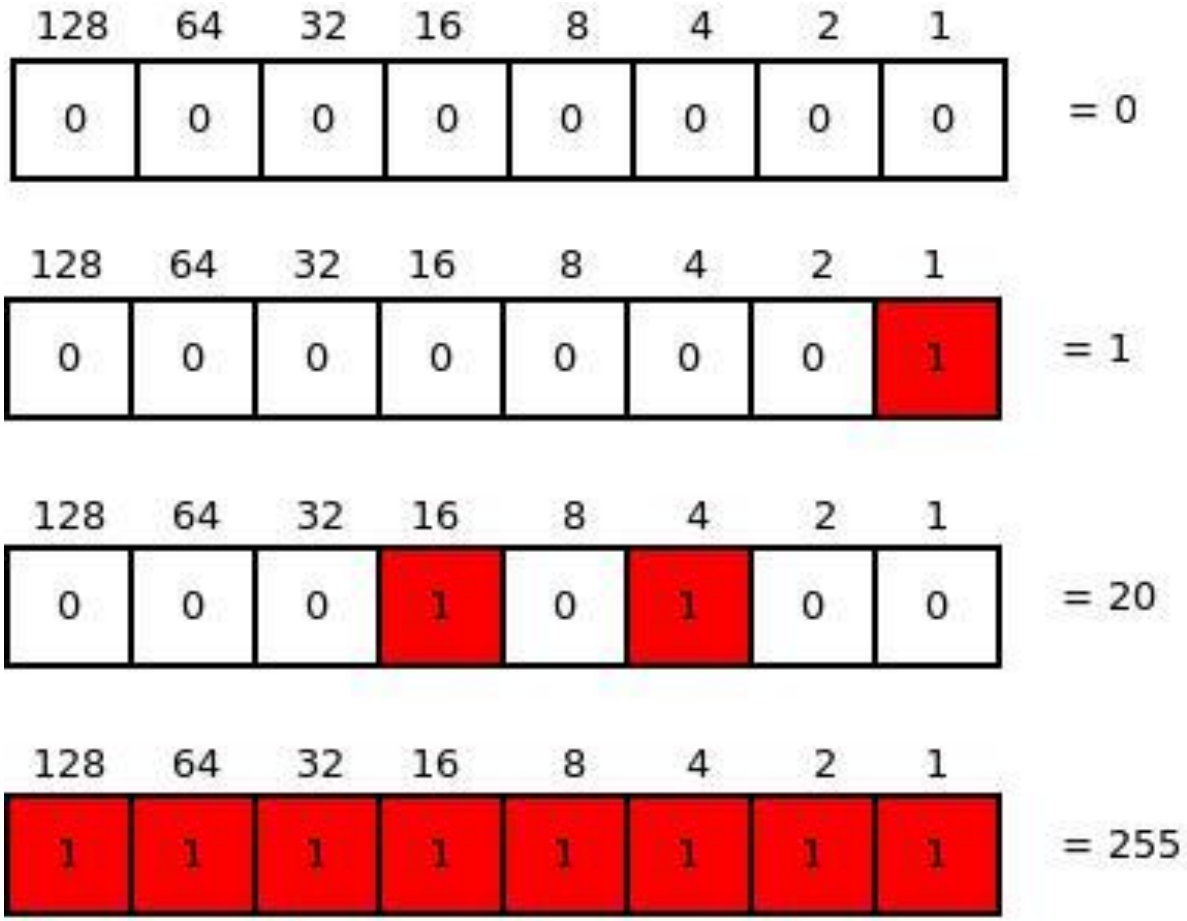
# Data Type Refresher

- Primitive data types (32bit):
  - signed char / unsigned char
  - signed short / unsigned short
  - signed int / unsigned int

- Redefinitions used for sizes, other things
  - size_t (unsigned)
  - ssize_t (signed)
  - etc

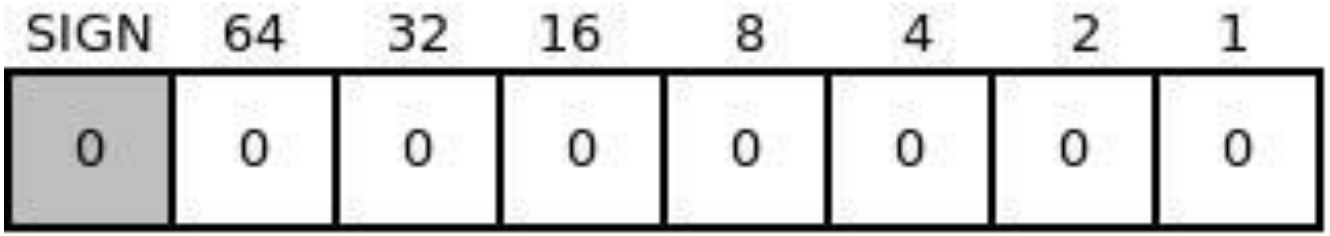
# Data Type Refresher

- 8bit unsigned char



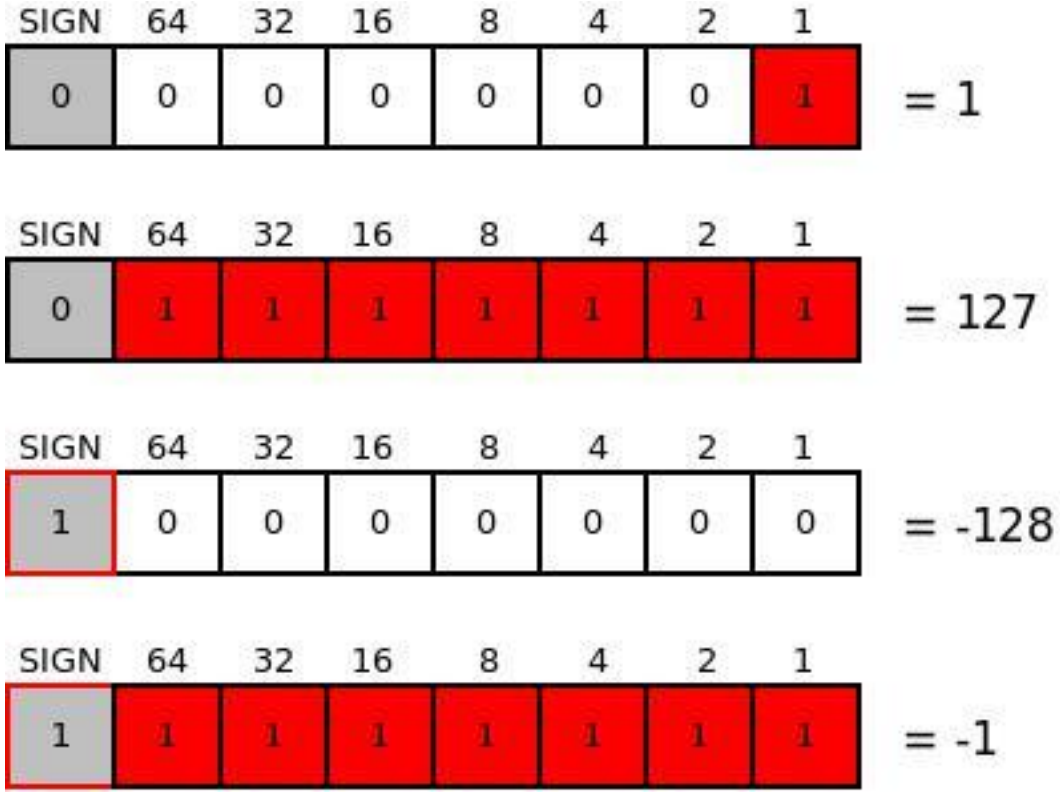
# Data Type Refresher

- Signed & Unsigned
  - Unsigned are used only to represent positive values
  - Signed can represent negative values
  - Signed data types use the highest bit as a "sign" bit, reflecting how the value should be treated



# Data Type Refresher

- When set, the sign bit reflects that the value is negative via Two's Compliment
- By default all primitive types are signed unless specifically declared unsigned

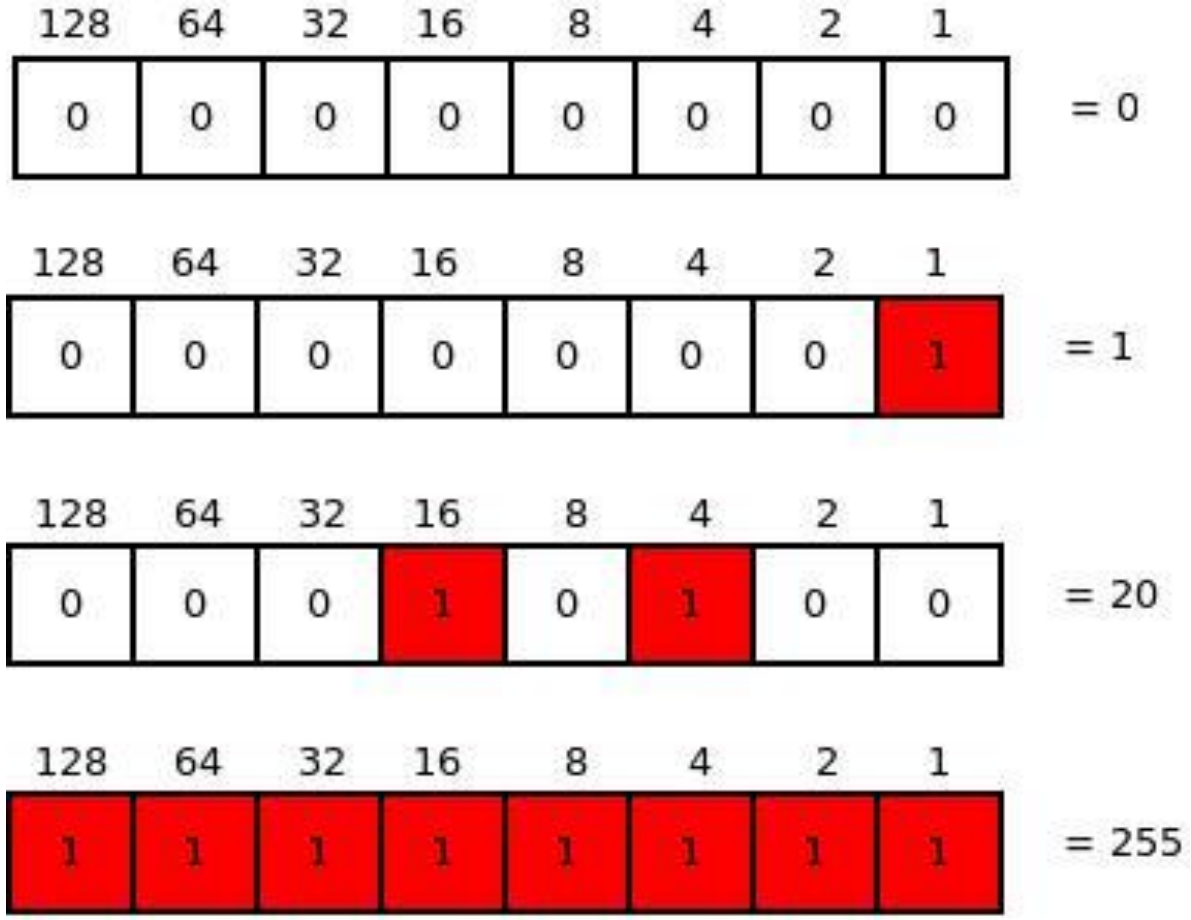


# Data Type Bugs

- Code becomes vulnerable when data types are used without consideration for the type boundaries
- For example, integers can contain a finite amount of data based on their bit-width (32bits)
- Exceeding this amount of data will result in the integer "wrapping"
- This phenomena is also called an "integer overflow".
- While not unique to integers, all data-type wrapping is often grouped into "integer overflows"

# Data Type Bugs

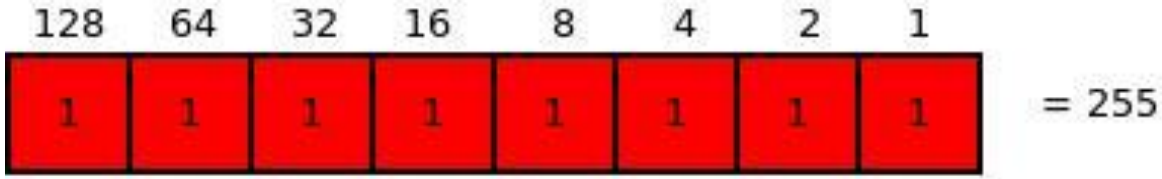
- Consider the 8bit example, what is the maximum value an 8bit unsigned data type can represent?



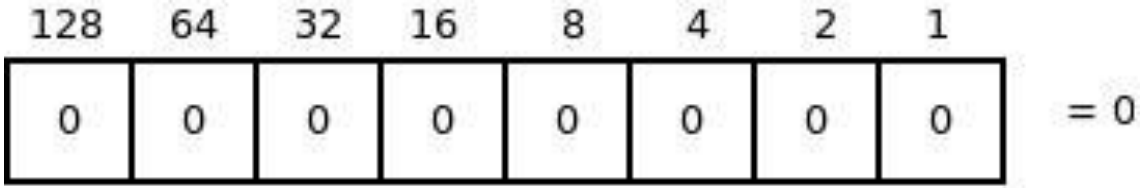
# Data Type Bugs

- What happens when you exceed this amount of data?

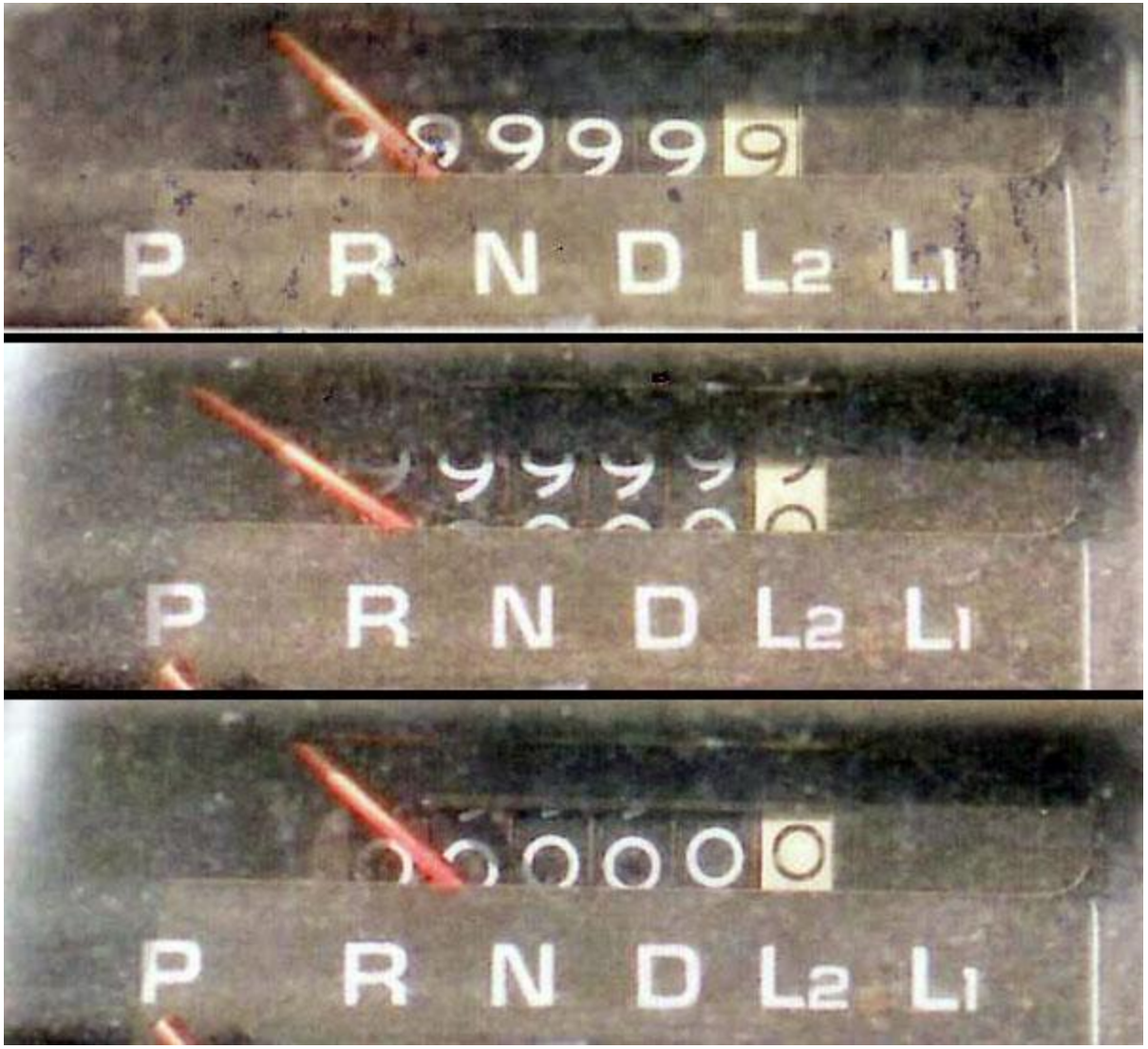
```
unsigned char x;
x = 255; /* maximum value */
```




```
x += 1;   /* what happens? */
          /* x is now 0! */
```



# Data Type Bugs



# Data Type Bugs

- Okay, so an integer can wrap. Now what?
- Consider the following simple code sample:

```
int getData(int sock)
{
    unsigned int len;
    char *buf = NULL;
    len = getDataLen(sock);
    buf = malloc(len + 1);
    read(sock, buf, len);
    buf[len+1] = 0x0;
 }
```

# Data Type Bugs

- The code intends to have enough space +1, potentially to store a NULL byte for a string
- If the network data supplied is `0xFFFFFFFF` (max 32bit value), when `1` is added, it will wrap to `0`
- This means the length passed to malloc() is zero bytes
- malloc() will thus return an under-sized buffer
- This allows for memory corruption during read()

# Data Type Bugs

- Pointers can wrap too…
- (Hint: pointers are secretly unsigned integers)

```
int StrStuff(int sock, char *buf, size_t buflen)
{
    size_t dataSize;
    char *maxpoint = buf + buflen;
    dataSize = readDataSize(sock);
    if (buf + dataSize < maxpoint)
    {
        read(sock, buf, dataSize);
        return 0;
    }
    return 1;
}
```

# Auditing Tips

- Review arithmetic used for size calculation
- Pay extra special attention
  - Dynamic memory size arithmetic
  - Pointer arithmetic used for size boundaries

- Quick and dirty: grep for malloc() and other memory allocation functions – check to see if sizes use arithmetic you can influence

# Questions?