Source Code Security Auditing and Vulnerabilities

CIS 4930 / 5930
Offensive Computer Security
Spring 2014

Outline of talk

- Intro
- CVE
- CCE
- CWE
- Strategy
- Common programming errors/bugs
- Source code auditing

Software Security Resources

See the:

- **Common Vulnerablities and Exposures**
  http://cve.mitre.org/
- Common Weakness Enumeration
  http://cwe.mitre.org/
- Seven kingdoms of weaknesses Taxonomy
  http://cwe.mitre.org/documents/sources/SevenPerniciousKingdomsTaxonomyGraphic.pdf
- Common Configuration Enumeration
  http://cce.mitre.org/

National Vulnerability Database

http://nvd.nist.gov/home.cfm

an example:
http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0861

CVEs (Common Vulnerabilities and Exposures)

- list of information security vulnerabilities that aims to provide common names for publicly known problems
- Goal is to make it easier to spread/share data
    - in house, between divisions, companies, researchers, etc.
    - across vulnerability databases
- Run by MITRE
- *should be taught in all software engineering classes....*

CVEs

- http://cve.mitre.org
- Intended to be a comprehensive list of publicly known vulnerabilities & exposures
- **vulnerability**: "is a mistake in software that can be directly used by a hacker to gain access to a system or network"
- **exposure**: "is a mistake in software that allows access to information or capabilities that can be used be a hacker as a stepping-stone into a system or network"

CCE (Common Configuration Enumeration)

- Assigns unique identifiers to configuration guidance statements
  - example **configuration guidance statements:**
    - The required permissions for accessing the directory %System Root%\System32\Setup should be "Administrator Account" only
    - The "account lockout threshold" for failed password attempts should be 3
    - For Linux, passwords should be stored in either SHA256 or SHA512, or the default DES formats and in the /etc/shadow file not the /etc/passwd file

CWE

A software **weakness** is an error that may lead to a software vulnerability, such as those enumerated by the CVE list

Examples software weaknesses include:

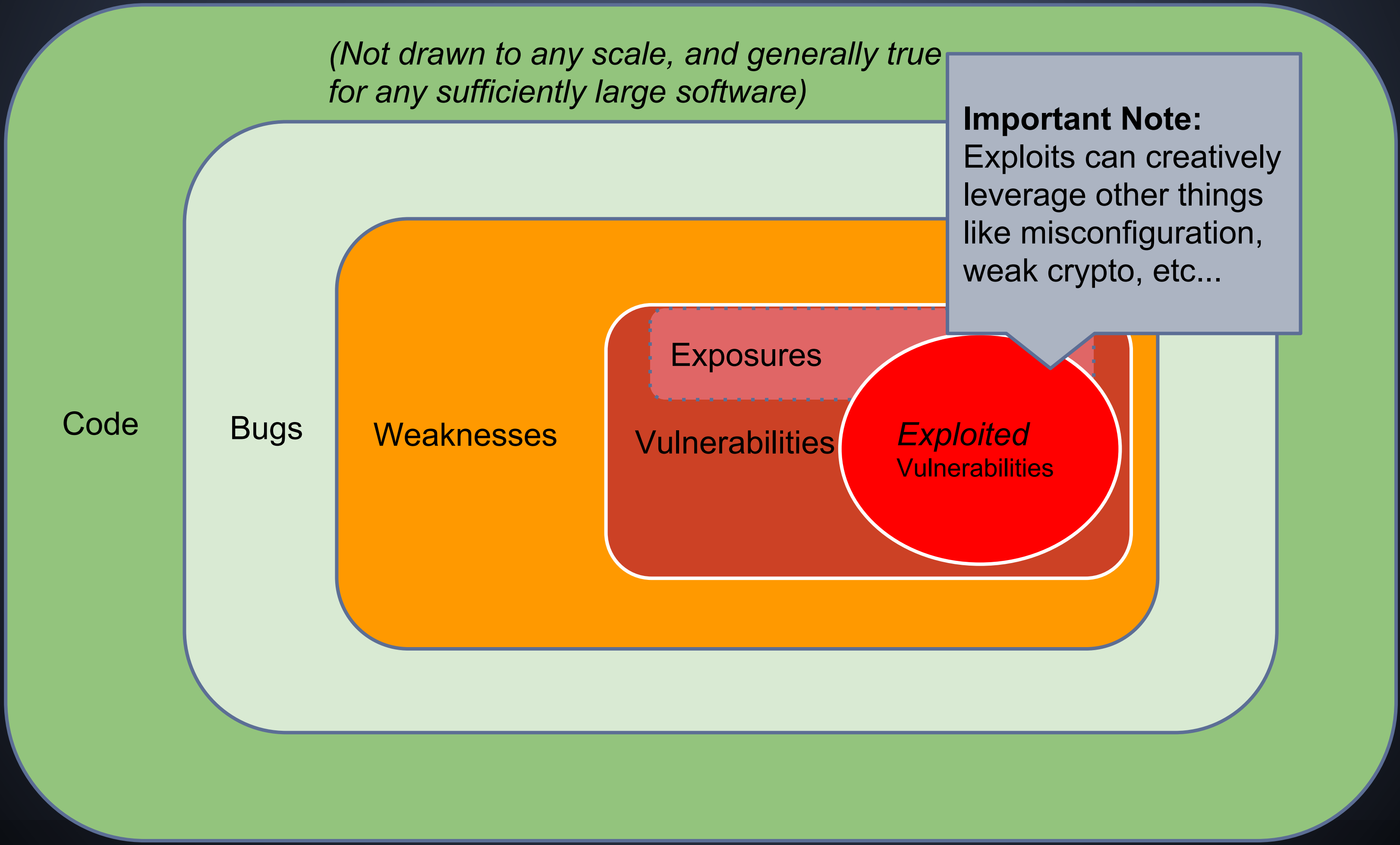
- buffer overflows, format strings, etc.
- structure and validity problems; common special element manipulations
- channel and path errors
- handler errors

More CWE Examples

- user interface errors
- pathname traversal and equivalence errors
- authentication errors
- resource management errors
- insufficient verification of data
- code evaluation and injection
- and randomness and predictability

Weaknesses are a subset of bugs

Code Nomenclature



(Not drawn to any scale, and generally true for any sufficiently large software)

Important Note: Exploits can creatively leverage other things like misconfiguration, weak crypto, etc...

Discovering Vulnerabilities

Three Primary Methods:

1. **Source Code Auditing**
    a. Requires source code
2. **Reverse Engineering**
    a. Can be done without source code.
    b. Requires binary applications (i.e. not interpreted languages)
    c. very time consuming and requires high technical skill
3. **Fuzzing**
    a. Lots of tools / frameworks exist
    b. Easy to make custom ones
    c. Binary or source code availability is unimportant

Source Code Auditing

- Tedious and time consuming
- Hard to estimate time cost
- Requires high knowledge/skill with given language

Source Code Auditing tools

- Author’s source code comments
- Editors / Reading tools
  - vi/vim; emacs; source-navigator; notepad++; eclipse; visual studio; Understand; source insight
- Pattern matching tools
- Static analyzers
  - prone to missing vulnerabilities
  - prone to false positives (can waste time)
- pen & paper
  - not obsolete yet

Approaches

- Find the most bugs?
- Find the easiest to find bugs?
- Find the weaknesses that are most reliable to exploit?

It is important to limit the approach

- won't ever have enough time to find all the bugs

[My] General Methodology

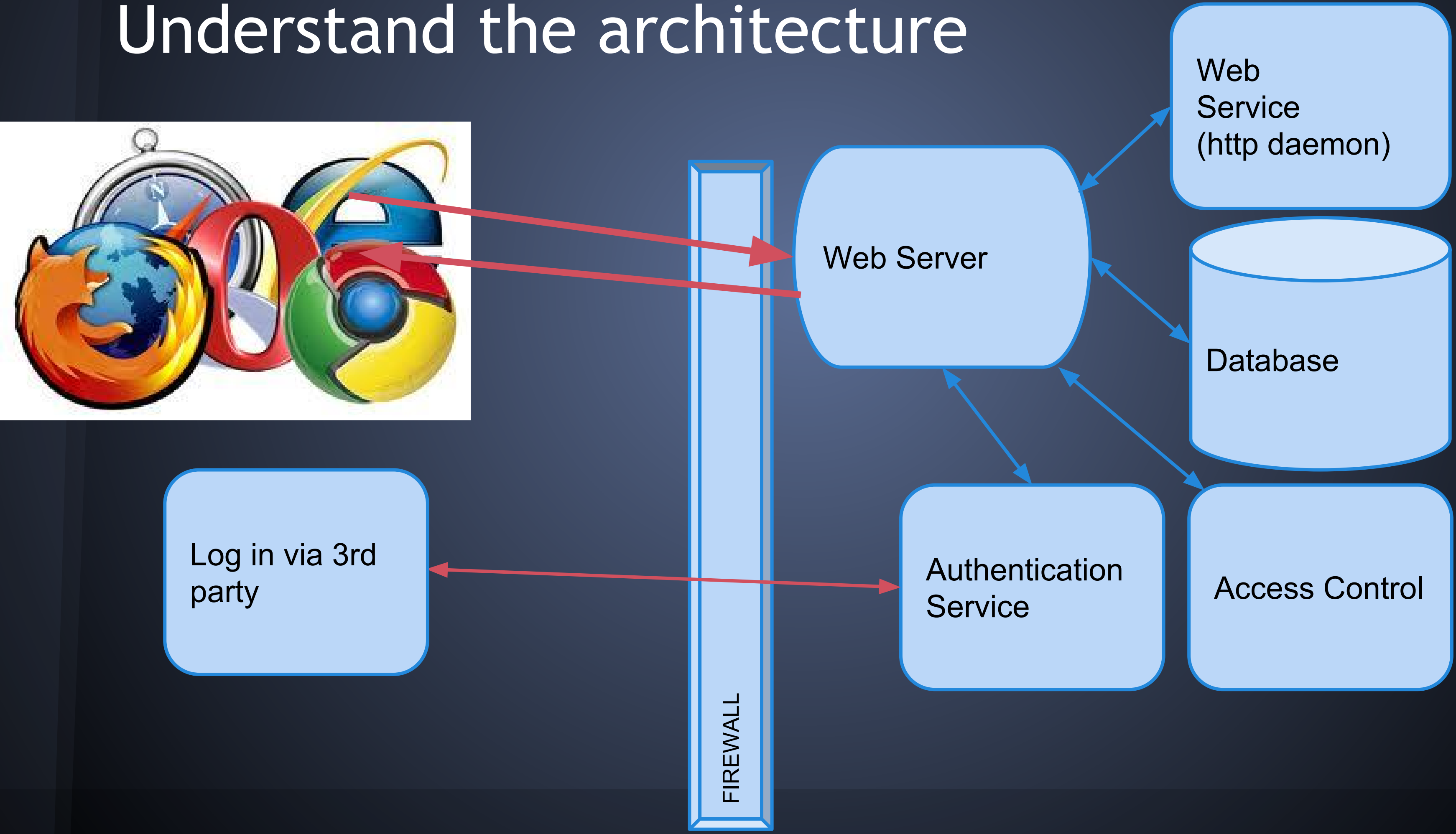
1. Understand the Application
    - features
    - architecture
    - programming language
2. Understand the Attack Surface
    - inputs
        i. various formats / protocols
    - code paths
3. Target your efforts
    - depends on your style

Understand the Application

- Read specs / documentation
- Understanding the programming language
  - Interpreted vs compiled
- Features
  - What features are really complex?
    - meld of two technologies or media encodings?
- Components
  - Database?
    - try to hit the Database for SQLi?
  - File share?
    - try to upload a file?

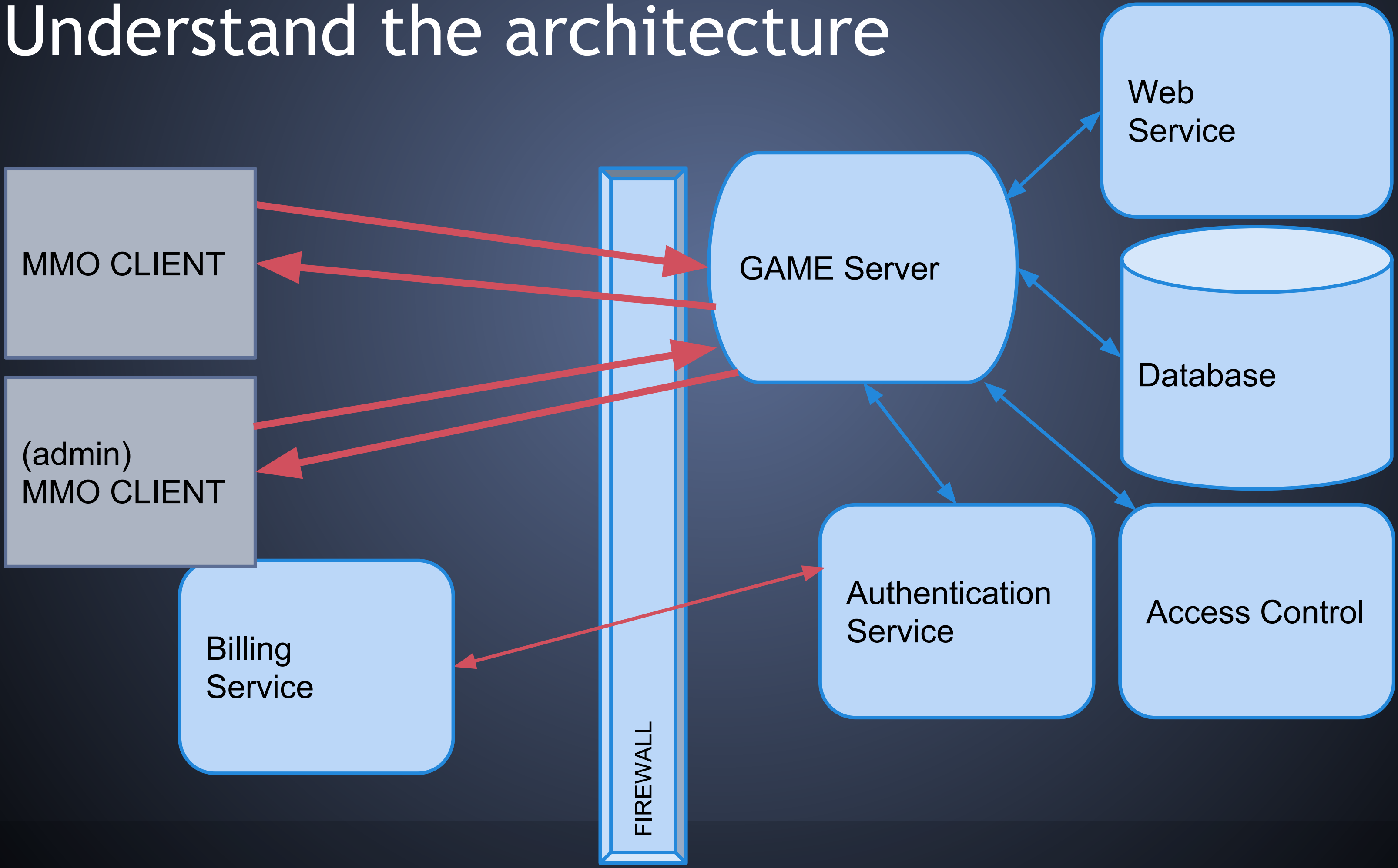
Understand the Application

Understand the architecture



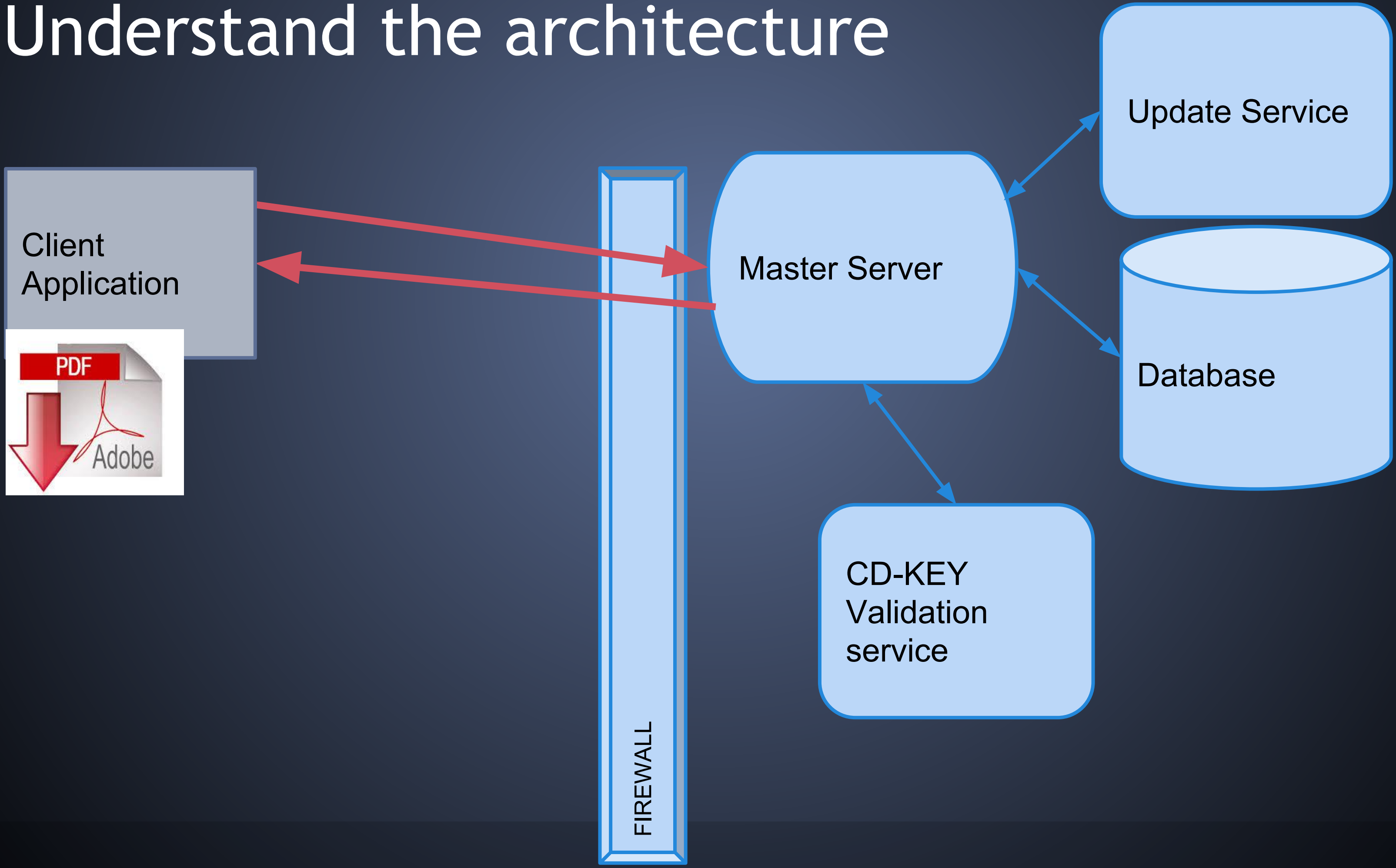
Understand the Application

Understand the architecture



Understand the Application

Understand the architecture



Understanding the Attack Surface

Attacker goals may vary

- You must choose which ones to focus on
    - sabotage?
        - defacing, attacker deleting records, altering them, destroying user trust
    - gaining access
        - to server/service
            - exploit free service?
        - of clients machines
            - attack / harass other users?
            - botnet?
            - identity theft?
    - piracy / theft

Understanding the Attack Surface

1. Understand inputs / outputs of architecture
   - dictates targets
2. Understand inputs of application(s)
   - dictates attack vectors for exploitation
3. Prioritize inputs of application that are remotely accessible
   - update()
   - sync()
4. Prioritize authentication mechanisms
   - weak cookies?
   - passwords sent in cleartext?
     i. plain encoding?

Understanding the Attack Surface

Remotely accessible code path:

- means functions / features that can be executed as a result (or following) network interaction / input

Remotely accessible code paths vs non:

- if code path is NOT remotely accessible, not likely to be remotely exploitable
  - read_config_file()
  - load_startup_scripts()
  - initialize()

Targeting:How to think like an attacker

Traditional strategies:

- input sources related to code paths
  - most effective
- target important components
  - Security Mechanisms
    - Authentication
    - http/https
  - Data managment / Database
  - Interpreters (php)
- Complex parsing, protocols, or functions

How to think like an attacker

"Meta Targeting" strategies

- Start by looking at source code comments
  - grep/search for:
    - FIX THIS, TODO!, XXXX, *******
    - Swearing / typos
    - old code
      - old libraries!
    - code checked in at 4AM
      - (its said that SSL was a largely a 4AM decision)
    - code checked in at same time as other buggy code
      - or patterns from other buggy code
      - or code from bad developers

Reading Code

- usually frustrating at times
- read iteratively
    - try to understand each component as you read it
        - gain a glimpse of the big picture
- skim past filler code
    - function prototypes
    - macros
    - initial or hardcoded value assignments

Reading Code

Tips from https://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review/

- Review fewer than 200-400 lines of code (LOC) at a time
  - significant diminishing returns above this
- Faster code review is not better
  - Optimal code review is around 300-500 LOC per hour
- Never review code for more than 90 mins at a time
  - significant diminishing returns after this

Reading Code

- Be Thorough
  - vast majority of code is OK
- Avoid making assumptions
  - can cause you to miss bugs, or assume something is done correctly (when it may not be)
-

Reading Code

- Look for abstraction
    - when used commonly, can be a big source for bugs
        - Look for when C++ style code and library calls break down into C style code / library calls
            - usually two developers from different backgrounds => bugs
    - misuses can lead to vulnerabilities
    - many devs love abstraction and use it as much as possible
- focus on code patterns
    - copy paste chunks
        - forgetting to tie up chunk variables i++, j++, k++, i++ ...

Quick review of topics from last time

- Integer signedness and promotion
- Format strings
- Off by one
- i++ vs ++i

Integer Signedness / Promotion

if (x > y)

- depends on x and y. if one is unsigned, will both be evaluated as unsigned

if (x > 16)

- 16 is signed by default. So if X is signed and is set to larger than MAX_INT, it will be negative

if (x > 16U)

- 16 Here is unsigned, so this will be safe due to promotion

Format Strings

printf(input);

- unsafe if input has conversion specifiers

printf(“%s”, input);

- safe, regardless if input has conversion specifiers

sprintf(tmp, “%s”, input)

printf(tmp)

- printf will be unsafe if tmp contains conversion specifiers

Off By One errors

```
char msg[5]
for (i = 0; i <= 5; i++)
    //use msg;
```

- should be < 5

Other example cases

- incrementing too many times
- improper calculation of bounds
- sizeof != strlen

i++ vs ++i

x = i++ - 5;

- will set x to i - 5, then increment i afterwards

x = ++i - 5;

- will increment i first, then set x accordingly

Programs in memory

When processes are loaded into memory, they are basically broken into many small sections

- .text Section
  - contains the machine instructions (read only)
- .data Section
  - global initialized variables
- .bss Section
- Heap Section
- Stack Section
- ...

General Bug Causes

- bugs in the way the code was implemented
  - can allow attackers to make the application behave in unintended ways
- main causes:
  - failure to validate input
  - programmer failure to understand an API
  - miscalculations
  - failure to validate results
    - of operations, functions, etc
  - application state failures

General Bug Causes

- other causes
    - Complex protocols
    - Complex file formats
    - Complex encoding / decoding / expansion
        - improper Unicode expansion (or other encoding)
    - Trusting the validity of input
    - failure to track relationships, object references, etc
        - look for where object-oriented-style, string-aware C++ code suddenly breaks down into C standard library calls.

Safe functions / API's

Despite the existence of safe functions and safe APIs

- Developers still misuse them or completely misunderstand them
    - improper calculation
        - of API inputs, of string size (forgot the NULL terminator)
    - improper parameters
        - the length variable can be completely misunderstood
    - etc..

Focusing on bugs that lead to vulnerabilities

less rambling, more usefulness

General Bug Categories -> vulns

Not complete list:

1. API Based Bugs
2. Programming Construct Errors
3. State Mechanics
4. External Resource Interactions
   - metacharacter injection

From: http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Wheeler-up.pdf

General Bug Cases

Not complete list:

- API Based Bugs
    - misuse of API’s provided by OS, language, or application
        - dangerous use of sprintf(), srncpy(), strncat(), printf(), syslog()...
        - overly complex APIs lead to dev errors

General Bug Cases

Not complete list:

- Programming Construct Errors
    - bad programming constructs
        - integer signedness
        - integer boundaries
        - logically wrong checks
        - bad boundary checks
        - using uninitialized vars / unchecked vars

General Bug Cases

Not complete list:

- State Mechanics
    - Bugs where process left in inconsistent state
        - thread safety issues
        - global variables
        - locks / deadlocks
        - privileges

General Bug Cases

Not complete list:

- External Resource Interactions
  - bugs where various components interact dangerously
    - SQLi ‘ “ ; --
    - XSS < >
    - directory traversal .. / .. /
    - special files (/dev/, LPT0, ...)

Metacharacter injection

- Different languages / interpreters have different metacharacters
- Often applications interface with other components
  - Sometimes these components are
    - shells
    - libraries / code in other languages
    - databases
- Important to note how each component handles metacharacters, and how bugs can be introduced

Metacharacter injection

Important cases

- comment symbols
  - -- in SQL
- union, or metacharacters that extend commands
  - &&, AND, ;
- wildcard symbols
  - *, %
- String closure/start
  - ' "

Integer overflow

source: pentest.cryptocity.net

```
int checkSize(unsigned int inputLength)
{
    unsigned short length;
    length = inputLength;
    if (length >= 128)
        return 1;
    return 0;
}
```

Integer overflow pt2

source: pentest.cryptocity.net

```
#define MAXSOCKBUF 4096
int readSocketData(int sock){
    char buf[MAXSOCKBUF];
    int length;
    read(sock, (char *)&length, 4);
    if (length < MAXSOCKBUF)
        read(sock,buf,length);
    //.....
}
```

//Comparison between two signed values

If length is 0xFFFFFFFF it will be -1 !

Send it a big packet and it will crash! (likely exploitable!)

Integer overflow pt2

source: pentest.cryptocity.net

```
#define MAXSOCKBUF 4096
int readSocketData(int sock){
    char buf[MAXSOCKBUF];
    int length;
    read(sock, (char *)&length, 4);
    if (length < MAXSOCKBUF)            //Comparison between two signed values
        read(sock,buf,length);
    //.....
}
```

So will read() still work? It only takes in unsigned ints for size parameter

Integer bug CVE-2001-0144

source: http://users.ece.cmu.edu/~dbrumley/pubs/integer-ndss-07.pdf

Can you spot it?

```
int detect_attack(u_char *buf, u_char *IV){
    static word16 *h = (word16 *) NULL;
    static word16 n = HASH_MIN_ENTRIES;
    register word32 i, j;
    word32 l;
    ...
    for (l=n; l<HASH_FACTOR(len/BSIZE); l=l<<2);
    if (h == NULL) {
        debug("Install crc attack detector");
        n = l;
        h = (word16 *)xmalloc(n*sizeof(word16));
    } //...
        for (c=buf, j=0; c < (buf+len); c+=BSIZE, j++){
            for(i=HASH(c) & (n-1); h[i] != UNUSED; i = (i+1) & (n - 1) .....
                h[i]   =   j;
        }
    }
```

Integer bug CVE-2001-0144

source: http://users.ece.cmu.edu/~dbrumley/pubs/integer-ndss-07.pdf

See: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2001-0144

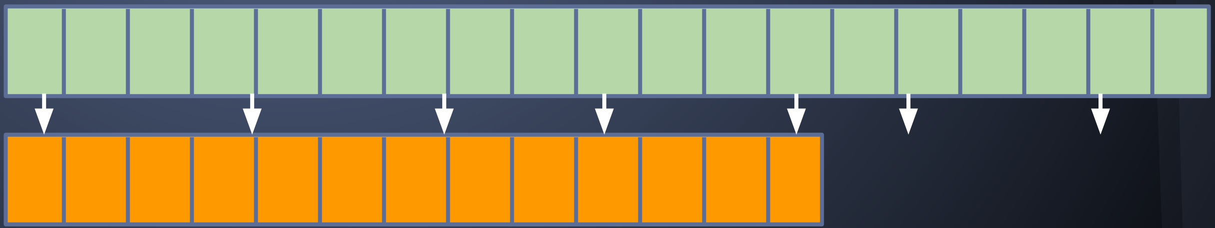
```
int detect_attack(u_char *buf, u_char *IV){
        static word16 *h = (word16 *) NULL;
        static word16 n = HASH_MIN_ENTRIES;
        register word32 i, j;
        word32 l;
        ...
        for (l=n; l<HASH_FACTOR(len/BSIZE); l=l<<2);
        if (h == NULL) {
                debug("Install crc attack detector");
                n = l;
                h = (word16 *)xmalloc(n*sizeof(word16));
        } //...
                for (c=buf, j=0; c < (buf+len); c+=BSIZE, j++){
                        for(i=HASH(c) & (n-1); h[i] != UNUSED; i = (i+1) & (n - 1) )
                                h[i]   =   j;
                }
        }
```

integer Truncation

Exploitable code that leads to memory corruption

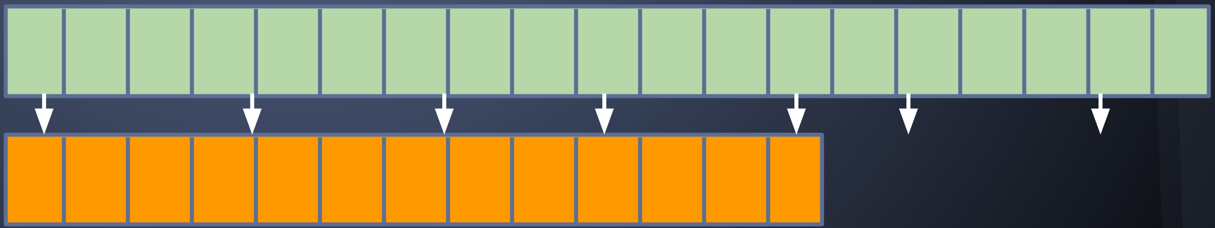
Buffer overflow

```
int some_function(char *inputstring)
{
    char buf[256];
    /* make a temp copy of data to work on */
    strcpy(buf, inputstring);
    .....

    return;
}
```



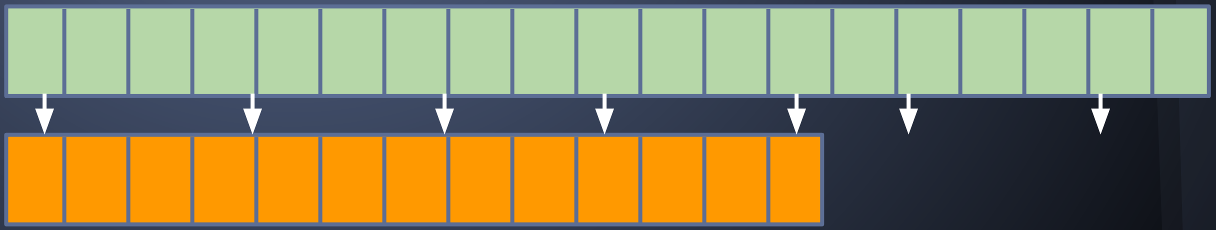
Buffer overflow pt2

```
int maybe_safer_function(char *inputstring)
{
    char buf[256];
    /* make a temp copy of data to work on */
    strncpy(buf, inputstring, strlen(inputstring));
    .....

    return;
}
```



Buffer overflow pt3

```
int maybe_safer_function(char *inputstring,
char * inputstring2)
{
    char buf[256];
    strncat(buf, inputstring, strlen(buf));
    strncat(buf, inputstring2, strlen(buf));
    .....
    return;
}
```



Now without hints

from http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Wheeler-up.pdf

CODE SURVEY

Programming Construct Error Example 5:

```
void bad_fn(char *input) {
  char buf[256], *ptr, *end, c;
  ptr = buf;
  end = &buf[sizeof(buf)-1];

  while(ptr != end) {
    c = *input++;
    if(!c)
      return;

    if(isalpha(c)) {
      *ptr++ = c;
      continue;
    }
    switch(c) {
      case ‘\\’:
        c = *input++;
        if(!c) return;
        *ptr++ = c;
        break;
      case ‘\n’:
        *ptr++ = ‘\r’;
        *ptr++ = ‘\n’;
        break;
      default:
        *ptr++ = c;
        break; }
  }// end while()
```

Example 2 (no hints, 3 vulns)

(from Jared DeMott. “Source Code Auditing”. Black Hat 2008.)

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

#define BUFLEN 16
int main(int argc, char *argv[]) {
  char buf1[16];
  char buf2[16];
  char buf3[BUFLEN];
  int i, len;

  if (argc != 12){
   exit(0);
  }
  strncpy(buf1, argv[1], sizeof(buf1));
  len = atoi(argv[2]);
    if (len < 16){
      memcpy(buf2, argv[3], len);
    } else
    {
      strcpy(buf2, "UNINITIALIZED");
      char *buf = malloc(len + 20);
      if (buf) {
        snprintf(buf, len+20,
        "String too long: %s", argv[3]);
        syslog(LOG_ERR, buf);
      }
    }

    // . . .
  }
```

Example 2.1 (no hints, 2 vulns)

(from Jared DeMott. “Source Code Auditing”. Black Hat 2008.)

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

int main(int argc, char *argv[]) {
  char buf3[BUFLEN];
  int i, len;
  char *buf4;
  char *buf5;
  char *buf6[16];

  if (argc != 12)
   exit(0);
  // . . .
  strncpy(buf3,argv[4], sizeof(buf3)-1);
  strncat(buf3, argv[5],sizeof(buf3)-1);
```

```
if (fork()){
  execl(“/bin/ls”, “/bin/ls”, argv[6], 0);
}
char *p; //filter out metacharacters
if (p =strchr(argv[7], ‘&’))
     *p = 0;
if (p =strchr(argv[7], ‘`’))
     *p = 0;
if (p =strchr(argv[7], ‘;’))
     *p = 0;
if (p =strchr(argv[7], ‘|’))
     *p = 0;
if (strlen(argv[7] > 1024){
     buf4 = malloc(20+strlen(argv[7]));
     sprintf(buf4, “/bin/cat %s”, argv
[7]);
     system(buf4);
```

Concluding Remarks

- Not Comprehensive coverage of bugs / types
- Best I expect you to understand without heavy C experience

Questions?