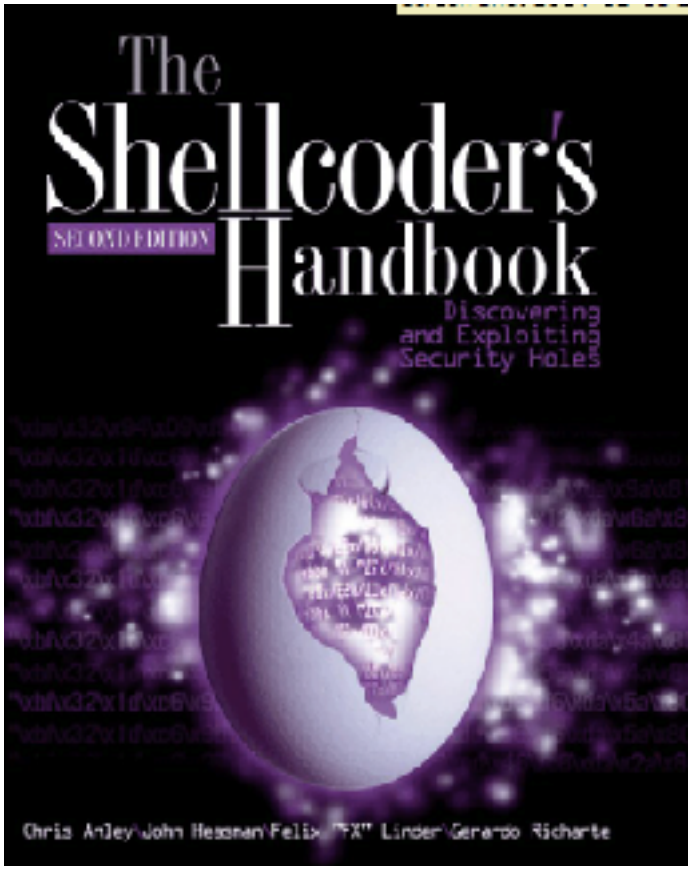
CNIT 127: Exploit Development

Ch 18: Source Code Auditing



Updated 4-10-17

Why Audit Source Code?

- Best way to discover vulnerabilities
- Can be done with just source code and grep
- Specialized tools make it much easier

Cscope

- A source code browsing tool
- Useful for large code trees, such as the whole Linux kernel
- Many useful search functions
- Cbrowser: GUI front-end
  - Links Ch 18a, 18b

Ctags

- Indexes source code
- Creates a tag file with locations for language tags in files scanned
- Works in many languages, including C and C++
  - Link Ch 18c

Text Editor

- Vim and Emacs have features that make writing and searching though large amounts of code easy
- Bracket-matching: find matching ([{

Automated Source Code Analysis Tools

Splint

Splint

Annotation-Assisted Lightweight Static Checking
Inexpensive Program Analysis Group
University of Virginia, Department of Computer Science

- Badly out-of date (last revised in 2007)
- Output a little hard to understand
  - Links Ch 18d, 18e

OPEN SOURCE STATIC CODE ANALYSIS SECURITY TOOLS

- Many available, specialized by language
- Link Ch 18f

Project 19x: Source Code Analysis with cppcheck (10 pts.)

- Easy to use
- Finds about half the obvious vulnerabilities we've exploited

Heap Overflow

GNU nano 2.2.6 File: heap0.c

```
{
 printf("level has not been passed\n");
}

int main(int argc, char **argv)
{
 struct data *d;
 struct fp *f;

 d = malloc(sizeof(struct data));
 f = malloc(sizeof(struct fp));
 f->fp = nowinner;

 printf("data is at %p, fp is at %p\n", d, f);

 strcpy(d->name, argv[1]);

 f->fp();
```

Finds Some Vulnerabilities

- But not the overflow!

```
root@kali:~/cppcheck# cppcheck heap0.c --enable=all
Checking heap0.c...
[heap0.c:40]: (error) Memory leak: d
[heap0.c:40]: (error) Memory leak: f
[heap0.c:34]: (error) Memory is allocated but not initialized: d
Checking usage of global functions..
[heap0.c:15]: (style) The function 'winner' is never used.
(information) Cppcheck cannot find all the include files (use --check-confi
g for details)
root@kali:~/cppcheck#
```

Format String Vulnerability

- It doesn't find it at all!

```
  GNU nano 2.2.6

#include <stdio.h>

int main(int argc, char **argv){
        char buf[1024];
        strcpy(buf, argv[1]);
        printf(buf);
        printf("\n");
}
```

```
root@kali:~/cppcheck# cppcheck fs.c --enable=all
Checking fs.c...
[fs.c:5]: (error) Buffer overrun possible for long command line
arguments.
Checking usage of global functions..
(information) Cppcheck cannot find all the include files (use --
check-config for details)
root@kali:~/cppcheck#
```

Flawfinder

- Much better
- In Kali
  - apt-get update
  - apt-get install flawfinder

Methodology

Top-Down (Specific) Approach

- Search for specific lines of vulnerable code, such as format string errors
- Auditor doesn't have to understand application in depth
- Misses vulnerabilities that span more than one part of the code

Bottom-Up Approach

- Auditor reads large portion of code
  - Starting at main()
- Time-consuming but can reveal subtle bugs

Selective Approach

- Most auditors use this approach
- Locate code that can be reached with attacker-defined input
  - Focus energy on that code
- Learn the purpose of that code thoroughly

Vulnerability Classes

Generic Logic Errors

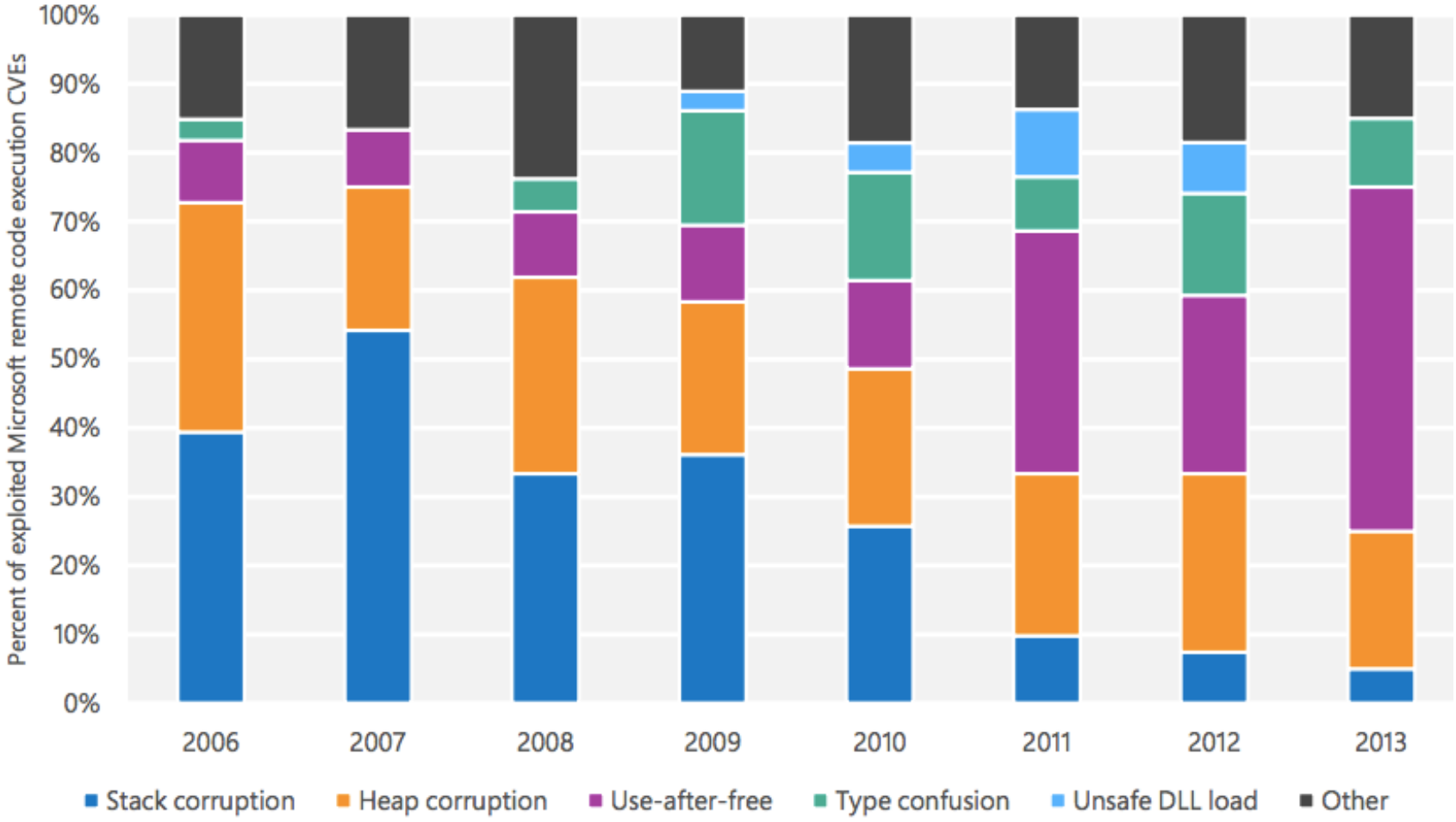
- Requires good understanding of an application
  - And internal structures and classes
- Example: wildcard certificates
  - Pascal-based CA will sell a certificate for ***\0.evil.com**
  - C-based browser will see it as *, a wildcard
    - Link Ch 18g

(Almost) Extinct Bug Classes

- Unbounded memory copy functions
  - strcpy(), sprintf(), strcat(), gets(), …
- Hunted nearly to extinction

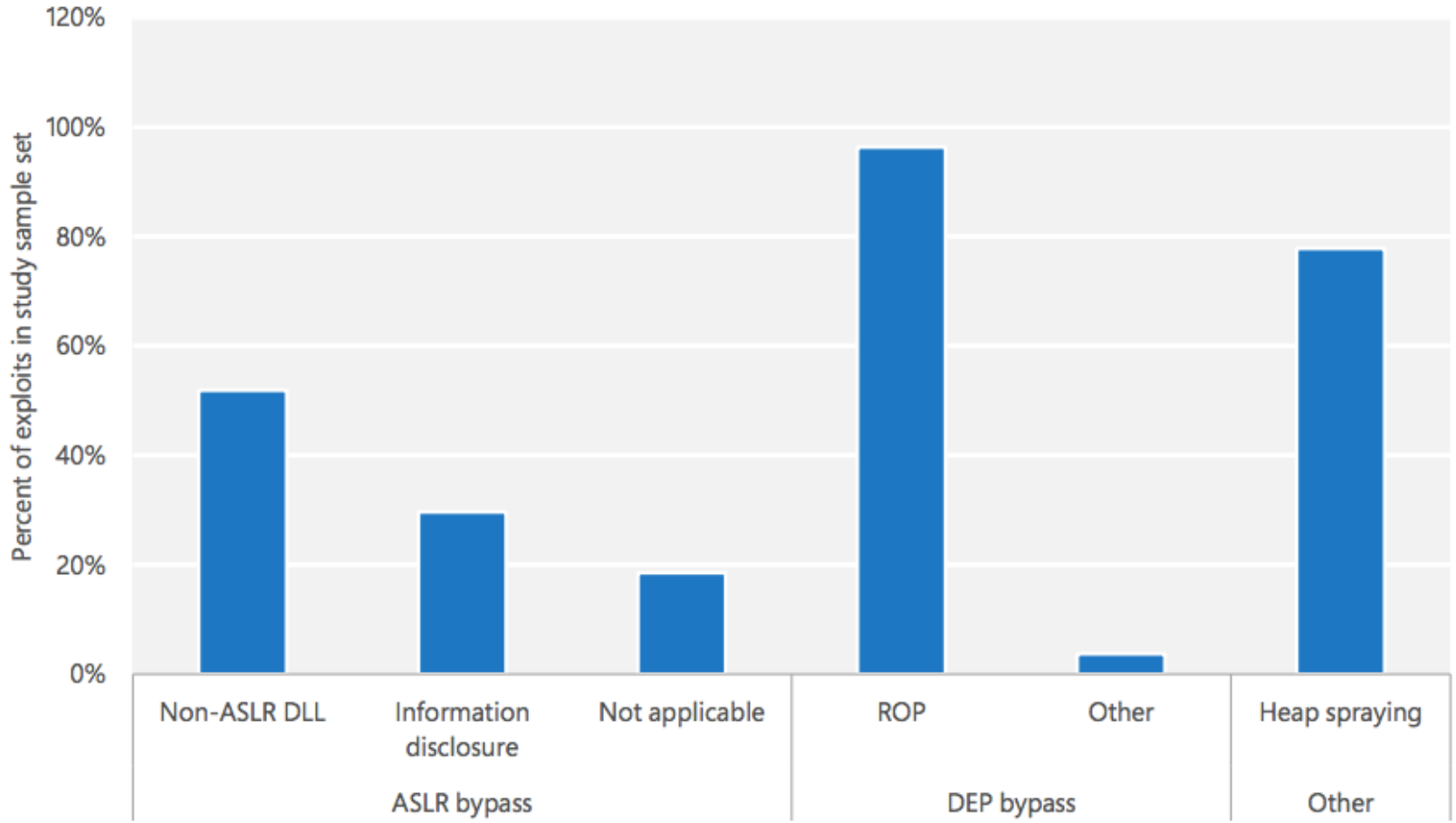
Root Cause (from Microsoft)

Figure 3. The root causes of exploited Microsoft remote code execution CVEs, by year of security bulletin



Bypassing ASLR & DEP

Figure 4. Techniques used by exploits targeting Microsoft products, January 2012–February 2014



Format Strings

- Easy to find with a code audit
  - Although cppcheck failed
- Often found in logging code
- Vulnerable only if attacker controls the format string

Possibly Vulnerable

```
syslog(LOG_ERR,string);
```

Non-Vulnerable

```
syslog(LOG_ERR,"%s",string);
```

Generic Incorrect Bounds-Checking

- Coder attempts to check limits, but does it incorrectly
- Example: Snort RCP Processor (2003)
  - Processes a series of RPC fragments
  - Checks each fragment to make sure it's not larger than the buffer
  - But it should check the total size of all combined fragments

Snort RCP Processor (2003)

```
while(index < end)
{
        /* get the fragment length (31 bits) and move the pointer to the
           start of the actual data */
        hdrptr = (int *) index;

        length = (int)(*hdrptr & 0x7FFFFFFF);

        if(length > size)
        {
            DebugMessage(DEBUG_FLOW, "WARNING: rpc_decode calculated bad "
                    "length: %d\n", length);
            return;
        }
        else
        {
            total_len += length;
            index += 4;
            for (i=0; i < length; i++,rpc++,index++,hdrptr++)
                *rpc = *index;
        }
}
```

Loop Constructs

- Coders often use intricate loops, and loops within loops
- Complex interactions can lead to insecurities
- Led to a buffer overflow in Sendmail

> The prescan() function in the address parser (parseaddr.c) in Sendmail before 8.12.9 does not properly handle certain conversions from char and int types, which can cause a length check to be disabled when Sendmail misinterprets an input value as a special "NOCHAR" control value, allowing attackers to cause a denial of service and possibly execute arbitrary code via a buffer overflow attack using messages, a different vulnerability than CVE-2002-1337.

- Link Ch 18h

Demonstration Exploit

- Link Ch 18i

```
[user@frida ~]$  sendmail user@diego -\

f"<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><
><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><
><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><
><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><
><><><><><><><>"
```

Off-by-One Vulnerabilities

- Often caused by improper null-termination of strings
- Frequently found in loops or introduced by common string functions
- Can lead to arbitrary code execution

Example from Apache

- When both **if** statements are true
  - Space allocated is one byte too small
  - memcpy will write one null out of bounds

```
 if (last_len + len > alloc_len) {
     char *fold_buf;
     alloc_len += alloc_len;
     if (last_len + len > alloc_len) {
        alloc_len = last_len + len;
     }
     fold_buf = (char *)apr_palloc(r->pool, alloc_len);
     memcpy(fold_buf, last_field, last_len);
     last_field = fold_buf;
 }
memcpy(last_field + last_len, field, len +1); /* +1 for nul */
```

OpenBSD ftp Daemon

- If last character is a quote, it can be written past the bounds of the input buffer

```
char npath[MAXPATHLEN];
    int i;

    for (i = 0; *name != '\0' && i < sizeof(npath) - 1; i++, name++) {
            npath[i] = *name;
            if (*name == '"')
                    npath[++i] = '"';
    }
    npath[i] = '\0';
```

strncat()

- Strncat always null-terminates its output string
- Will write a null byte out of bounds unless the third argument is equal to the remaining space in the buffer minus one byte

The following example shows incorrect usage of `strncat`:

```
strcpy(buf,"Test:");
strncat(buf,input,sizeof(buf)-strlen(buf));
```

The safe usage would be:

```
strncat(buf,input,sizeof(buf)-strlen(buf)-1);
```

Non-Null Termination Issues

- If a string is not terminated with a null
  - Memory after the string is interpreted as part of the string
  - May increase length of string
  - String writes may corrupt memory outside the string buffer
  - Can lead to arbitrary code execution

strncpy()

- If there's not enough space in the destination buffer
  - strncpy() won't null-terminate the string it writes

strncpy() Example

- First strncpy won't null-terminate not_term_buf
- Second strcpy is unsafe, even though both buffers are the same size

```
char dest_buf[256];
char not_term_buf[256];

strncpy(not_term_buf,input,sizeof(non_term_buf));

strcpy(dest_buf,not_term_buf);
```

- Fix it by adding this line of code after the first strcpy

```
not_term_buf[sizeof(not_term_buf) - 1] = 0;
```

Skipping Past Null-Termination

- String-processing loops that process more than one character at a time
  - Or where assumptions about string length are made
- Can make it possible to write past end of a buffer
  - Possible arbitrary code execution

Example from Apache

- This line is intended to skip past **://** in a URL
  - cp += 3

```
else if (is_absolute_uri(r->filename)) {
    /* it was finally rewritten to a remote URL */

    /* skip 'scheme:' */
    for (cp = r->filename; *cp != ':' && *cp != '\0'; cp++)
        ;
    /* skip '://' */
    cp += 3;
```

But Not All Schemes End in ://

- If the URI is **ldap:a**
  - The null byte is skipped

```
int i = strlen(uri);
if (    (i > 7 && strncasecmp(uri, "http://",   7) == 0)
     || (i > 8 && strncasecmp(uri, "https://",  8) == 0)
     || (i > 9 && strncasecmp(uri, "gopher://", 9) == 0)
     || (i > 6 && strncasecmp(uri, "ftp://",    6) == 0)
     || (i > 5 && strncasecmp(uri, "ldap:",     5) == 0)
     || (i > 5 && strncasecmp(uri, "news:",     5) == 0)
     || (i > 7 && strncasecmp(uri, "mailto:",   7) == 0) ) {
     return 1;
}
else {
     return 0;
}
```

Signed Comparison Vulnerabilities

- Coder attempts to check input length
- But uses a signed integer variable
- Or two different integer types or sizes
  - C sometimes converts them both to signed integers before comparing them
- Following example from Apache
  - Led to code execution on Windows and BSD Unix

Example from Apache

- **bufsize** is a signed integer
  - Remaining space in the buffer
- **r->remaining** is signed
  - Chunk size from the request
- **len_to_read** should be the smaller of the two
  - Negative chunk size tricks the code into performing a large memcpy later, because it's cast to unsigned

```
len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;

len_read = ap_bread(r->connection->client, buffer, len_to_read);
```

Integer Conversions

Source Size/Type	Source Value	Destination Size/Type	Destination Value
16-bit signed	–1 (0xffff)	32-bit unsigned	4294967295 (0xffffffff)
16-bit signed	–1 (0xffff)	32-bit signed	–1 (0xffffffff)
16-bit unsigned	65535 (0xffff)	32-bit unsigned	65535 (0xffff)
16-bit unsigned	65535 (0xffff)	32-bit signed	65535 (0xffff)
32-bit signed	–1 (0xffffffff)	16-bit unsigned	65535 (0xffff)
32-bit signed	–1 (0xffffffff)	16-bit signed	–1 (0xffff)
32-bit unsigned	32768 (0x8000)	16-bit unsigned	32768 (0x8000)
32-bit unsigned	32768 (0x8000)	16-bit signed	–32768 (0x8000)
32-bit signed	–40960 (0xffff6000)	16-bit signed	24576 (0x6000)

How to Create a Secure Login Script in PHP and MySQL

```
function login($email, $password, $mysqli) {
```

```
 // hash the password with the unique salt.
 $password = hash('sha512', $password . $salt);
```

```
// Check if the password in the database matches
// the password the user submitted.
if ($db_password == $password) {
    // Password is correct!
```

- Link Ch 18l

PHP Hash Comparison Weakness A Threat To Websites, Researcher Says

Flaw could allow attackers to compromise user accounts, WhiteHat Security's Robert Hansen -- aka "RSnake" -- says in new finding on 'Magic Hash' vulnerability.

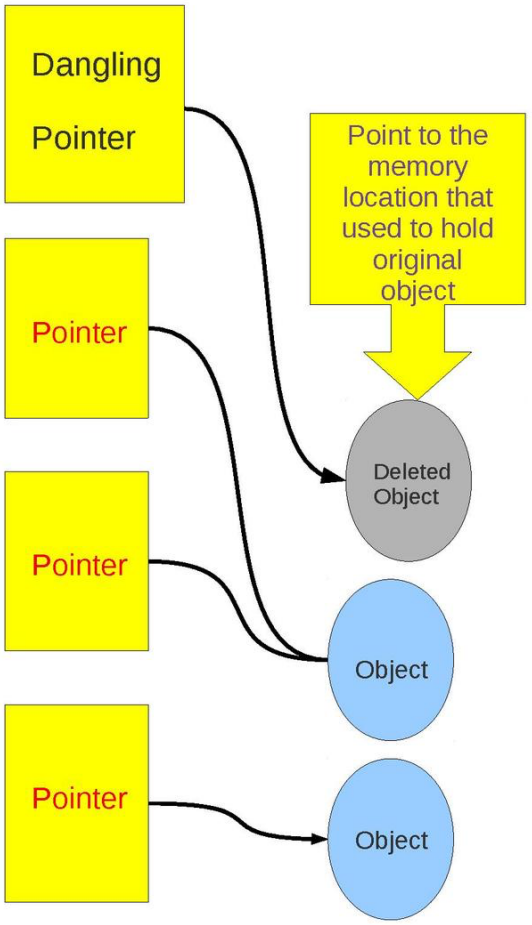
- A hashed password can begin with 0e and contain only digits (very rare)
  - Like 0e123535896618210356 85
- PHP reads that as scientific notation
  - 0^123…
  - Always zero (link Ch 18j)

Double Free Vulnerabilities

- Freeing the same memory chunk twice
- Can lead to memory corruption and arbitrary code execution
- Most common when heap buffers are stored in pointers with global scope
  - Good practice: when a global pointer is freed, set it to Null to prevent it being re-used
  - Prevents dangling pointers

Out-of-Scope Memory Usage Vulnerabilities

- Use of a memory region before or after it is valid
- Also called "Dangling Pointer"
  - Image from Wikipedia
    - Link Ch 18k)



Uninitialized Variable Usage

- Static memory in the **.data** or **.bss** sections of an executable are initialized to null on program startup
- But memory on the stack or heap is not
- Uninitializes variables will contain data from previous function calls
  - Argument data, saved registers, or local variables from previous function calls

Uninitialized Variable Usage

- Rare, because they can lead to immediate program crashes
  - So they get fixed
- Look for them in code that is rarely used
  - Such as handlers for uncommon errors
- Compilers attempt to prevent these errors

Example

- If data is null
  - test is never assigned any value
  - But test is still freed

```
int vuln_fn(char *data,int some_int) {
        char *test;

        if(data) {
                test = malloc(strlen(data) + 1);
                strcpy(test,data);
                some_function(test);
        }

        if(some_int < 0) {
                free(test);
                return -1;
        }

        free(test);
        return 0;
}
```

Exploitation

- The "uninitialized" data in test is not random
- It comes from previous variables and function calls
- It may be controlled by the attacker
- So the free() leads to a controllable memory write
  - Arbitrary code execution

Use After Free Vulnerabilities

- Heap buffers are temporary
  - Released with free()
- But a program may use a pointer after free()
  - If more than one variable points to the same object
- Allows an attacker to write to RAM
  - Possible arbitrary code execution

Multithreaded Issues and Re-Entrant Safe Code

- A global variable is used by more than one thread, without proper locking
  - A variable might be changed unexpectedly by another thread
- Such issues won't appear until the server is under heavy load
  - May remain as intermittent software bugs that are never verified