Advanced Software Vulnerability Assessment

The art and science of uncovering subtle flaws in complex software…

Neel Mehta <nmehta@iss.net>

Mark Dowd <mdowd@iss.net>

Chris Spencer <cspencer@iss.net>

Halvar Flake <halvar@blackhat.com>

Nishad Herath <nherath@iss.net>

Overview

Multitude of contemporary documentation addresses security vulnerability exploitation in depth, yet none address the methodology and techniques of vulnerability assessment thoroughly, if at all.

- Vulnerability classes and identification.
- Methodology of assessment.
- Tools of the trade.

Topics

- Dangerous library/API functions.
- Unsafe use of pointer arithmetic.
- Subtle trust relationships.
- Integer manipulation issues.

Methodology

- Top down methodology.
- Bottom up methodology.
- Hybrid approach.

Top down approach

Start at the entry point and follow all code paths.

Pros:

- Complete coverage of the codebase.
- In depth understanding of the application functionality.

Cons:

- Very tedious and time consuming.
- Sometimes not feasible.
- Potential waste of resources.

Bottom up approach

Pros:

- Can potentially uncover bugs quickly.
- Better utilization of resources.

Cons:

- Potential to miss subtle issues and issues with wider coverage.
- Usually, an in-depth understanding of the application functionality is not achieved.
- Potential to miss code paths that might have had issues.

Hybrid approach

- Incorporates elements from both top down and bottom up methodologies effectively according to the requirements.
- Attempts to gain the advantages offered by the pervious methods.
- Attempts to maximize results with a minimal amount of resources.
- Streamlines the process and eliminates of reduces the impact of the disadvantages of the previous approaches.
- Targets the critical code paths and analyzes them in depth while still maintaining a sufficient level of overall analysis.

Tracking

- Tracking execution states is a helpful addition to the methodologies described.
- Requirements definition.
- Desk checking.
- Following and reverse engineering the programmers logic and often, making educated guesses about the programmer’s style of thinking.

Tools

- Editors
- Source Browsers
- Automated auditors
- Miscellaneous

Editors

- VIM:

Syntax highlighting

Bracket matching

Tags

- EMACS
- PICO

Source Browsers

- Source Navigator
- Cscope:

1. Recursively search
2. Find any symbol definition, or use.
3. Function calls, or functions called.
4. Plugins

- Cbrowser

Automated Auditing Tools

- ITS4
- SPLINT
- Cqual

Miscellaneous

- CVSWeb

Dangerous functions.

- Unbounded memory copies such as strcpy(), strcat() etc.
- Bounded memory copy functions.

Bounded memory copy functions.

- strncpy() NULL termination problems.
- Misleading size value in strncat().
- strncat() doesn’t account for the trailing NULL.
- Potential underflow problems when addressing the trailing NULL issue.
- Misuse of return values in *snprintf().

Remaining length issue.

```
char buf[1024];

…
strcpy(buf, “user entered: “);
strncat(buf, user_data, sizeof(buf));
…
```

Off-by-one.

```
char buf[1024];

…
strcpy(buf, “user entered: “);
strncat(buf, user_data, sizeof(buf) – strlen(buf));
…
```

Underflow issue.

```
char buf[1024];

…
strncpy(buf, user_data);
strncat(buf, sizeof(buf) – strlen(buf) – 1, more_data);
…
```

Pointer Arithmetic.

- Looping Constructs.
- Miscalculations.
- Off-by one errors.

Looping constructs: ntpd

```
while (cp < reqend && isspace(*cp))
       cp++;
   if (cp == reqend || *cp == ',') {
       buf[0] = '\0';
       *data = buf;
       if (cp < reqend)
           cp++;
       reqpt = cp;
       return v;}
   if (*cp == '=') {
       cp++;
       tp = buf;
       while (cp < reqend && isspace(*cp))
           cp++;
       while (cp < reqend && *cp != ',')
           *tp++ = *cp++;                 // here is the problem
    if (cp < reqend)
           cp++;
       *tp = '\0';
       while (isspace(*(tp-1)))
           *(--tp) = '\0';
       reqpt = cp;
       *data = buf;
       return v;
   }
```

Looping constructs: lpd

```
if ((tmp = strtok(NULL, "\n\t ")) != NULL) {
                        .. does stuff ..
            }
            current_request = tmp_id;
            tmp = NULL;

            switch (buf[0]) {
            ...
            does stuff here
            ...
            case '\3':  /* Transfer Data File */
                        ...
                        df_list[file_no++] = strdup(name);
                        ACK(ofp);
                        break;
```

Miscalculations.

- Sometimes when performing pointer arithmetic, miscalculations can be made as to how much space is left in a buffer. This can allow for malicious attackers to sometimes write outside the bounds of the destination buffer.
- Subtle and difficult to detect.
- For example, the BIND 8 TSIG issue discovered last year.

Off-by-one: OpenBSD ftpd.

```
void
replydirname(name, message)
        const char *name, *message;
{
        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';
        reply(257, "\"%s\" %s", npath, message);
}
```

Off-by-one miscalculation.

```
char buf[1024], *ptr = buf;
int len  = sizeof(buf);

buf[1023] = ‘\0’;
strncpy(ptr, user_data, len - 1);
ptr += strlen(ptr) + 1;
len = &buf[sizeof(buf)-1] – ptr;
strncpy(ptr, more_user_data, len);
```

Excessive pointer increment.

```
char buf[1024], *ptr = buf;

ptr += snprintf(ptr, sizeof(buf), “%s”, user_data);
ptr += snprintf(ptr, sizeof(buf)-(ptr – buf), “%s”, more_data);
```

Union Mismanagement

- Basics: What is a union?

```
union {
   int an_integer;
   short a_short_integer;
   char *a_pointer;
} u;
```

Security Implications

- Mistaking one data member for another
- Can occur in input processing routines
- Type Confusion

Subtle trust relationships

- Some subtle trust relationships exist between internal functional units of software systems.
- Server applications at times trust externally available information that is indirectly related to the intended functionality or infrastructure services which the client controls.
- Server applications assume the intended client software is the only client software that will access the server services.
- In case of application errors, applications tend to trust the integrity of it’s own address space at the point of graceful termination or saving logging information.

Internal trust relationships

- Many major operating systems contain undocumented internal APIs that assume obscure API parameters to be consistent between the client and server subsystems. However, since the client subsystems are directly modifiable by the client, the trust can be broken and the server subsystem can be exploited.
- Shared codebase, shared development resources and the same programmer mindset leads to implicit assumptions and its only human to make assumptions.

Trusting external information

- Servers often trust indirectly related information about the clients, where the information happens to be in control of the client.
- For example, Microsoft Exchange Server 5.5 trusts the client IP address to resolve to a DNS name with certain properties as most often the case with legitimate real life clients.
- In the recently released advisory, its demonstrated how a client could violate this implicit trust (or assumption on the part of the server) to exploit the server and take complete control over the server.

Specific client assumption.

- Many servers that utilize proprietary protocols assume that the specific client software intended to be used with the server application will be used to communicate to the server.
- Many assumptions or “mutual understandings” exist which leads to non-thorough security coverage and unsafe programming practices which could be exploited.
- For example, the Oracle TNS listener DoS and buffer overflow vulnerabilities discovered last year.

Is safe really safe?

- Attempts to be safe and robust which are not very well researched and tested leads to being unsafe and vulnerable.
- When an application handles an error (signal handlers, exception handlers, exit routines, logging functionality) and performs tasks after the error occurred, it is making an assumption of the relative integrity of its own address space, which is very ignorant to say the least.
- Ideally, post-error processing should be handled by an out-of-process address space entities on behalf of the application that the fault occurred in.

Integer manipulation

- Signed and unsigned integers
- Different sized integers
- Integer wrapping

Introduction

- Usually 32 bit
- Signed unless explicitly qualified with the ‘unsigned’ keyword

	16-bit signed	16-bit unsigned	32-bit signed	32-bit unsigned	64-bit signed	64-bit unsigned
min	-32768	0	-2147483648	0	-9223372036854775808	0
max	32767	65535	2147483647	4294967295	9223372036854775807	18446744073709551615

Signed/Unsigned Issues

- Negative integers for length specifiers can occur if either the user supplies an unchecked length, or a calculation is made based on an unexpected value
- Can be used to bypass maximum length restrictions

Signed Bug Example

```
#define MAX_LEN 256
….
int datalen;
char buf[MAX_LEN];

datalen = get_int_from_socket();

if(datalen > MAX_LEN){
    printf(“invalid data sent – data field too large”);
    exit(-1);
}

if(read(sock, buf, datalen) < 0){
    perror(“read”);
    exit(errno);
}
```

Different Sized Integers

- Integers of different sizes are sometimes used during calculation
- Implications of this could include truncated values (32 bit -> 16 bit), sign issues (32 bit -> 16 bit or vice versa)

Integer wrapping

- Causing an integer to exceed maximum boundary value, or decrease below the minimum value

Example 1: Addition

```
unsigned int len;
char *str;

…

len = get_user_length();

if(!(str = (char *)malloc(len + 1))){
    perror(“malloc”);
    exit(errno);
}

memcpy(str, user_data, len);
…
```

Example 2: Subtraction

```
#define HEADER_SIZE 32
#define MAX_PACKET 256

int len;
char buf[MAX_PACKET], data[MAX_PACKET];

…
if((len = read(sock, buf, sizeof(buf)-1)) < 0){
    perror(“read”);
    exit(errno);
}

memcpy(data, buf+HEADER_SIZE, len-HEADER_SIZE);
```

Example 3: Multiplication

```
int num, i;
object_t *objs;
…

num = get_user_num();

if(!(objs = (object_t *)malloc(num * sizeof(object_t)))){
    perror(“malloc”);
    exit(errno);
}

for(i = 0; i < num; i++){
    objs[i] = get_user_object();
}
```

Problems

- Often there are problems involved in exploiting such bugs
- Large data copies often occur, or
- Large amounts of data need to be supplied (eg. strdup() type functions)

Solutions

- Catching Segmentation Violations
- Delivery of other caught signals
- Structured Exception Handling (Win32)
- Cleanup routines (deallocation of objects)

Best Case Scenario

- Integer is directly controlled by the user
- A loop is entered for the data copy that can be prematurely terminated
- Premature termination of loop does not result in immediate call to exit()

Integer Wrapping thoughts

- The potential of exploitation of these types of bugs has not been fully realised
- Applies to pointer types as well
- Certain functions can be dangerous – calloc(), new operator in C++ in some implementations

Real Example 1: OpenSSH

- Challenge-Response bug perfect example of multiplication integer overflow
- Overflow causes incorrect data allocation on the heap
- We write past the end of the allocated data with pointers returned from packet_get_string

OpenSSH Example cont’d

- We can write over a fatal_cleanup function pointer with the pointer returned from packet_get_string()
- Premature termination of loop receiving input calls fatal(), which calls the function pointer which we overwrite

xdr_array() example:

```
c = *sizep;
if ((c > maxsize) && (xdrs->x_op != XDR_FREE)) {
        return (FALSE);
}
nodesize = c * elsize;
if (target == NULL)
        switch (xdrs->x_op) {
        case XDR_DECODE:
                if (c == 0)
                        return (TRUE);
                *addrp = target = mem_alloc(nodesize);
                ...
                break;
                ...
}
for (i = 0; (i < c) && stat; i++) {
        stat = (*elproc)(xdrs, target);
        target += elsize;
}
```