Vulnerability Assessment: Code Auditing

Study Slides

Methodology – Why Is It Important?

- Game-plan for attack
- Coverage: emphasize on high-prize target code
- Time: never enough, make the the best of it
- Reporting:
  - Professional assessment is more than just busticating code
  - Writing reports, often intended for a non-or-less technical audience
  - Potentially writing extensive remediation documentation or patches
  - Clients love to hear about the business process

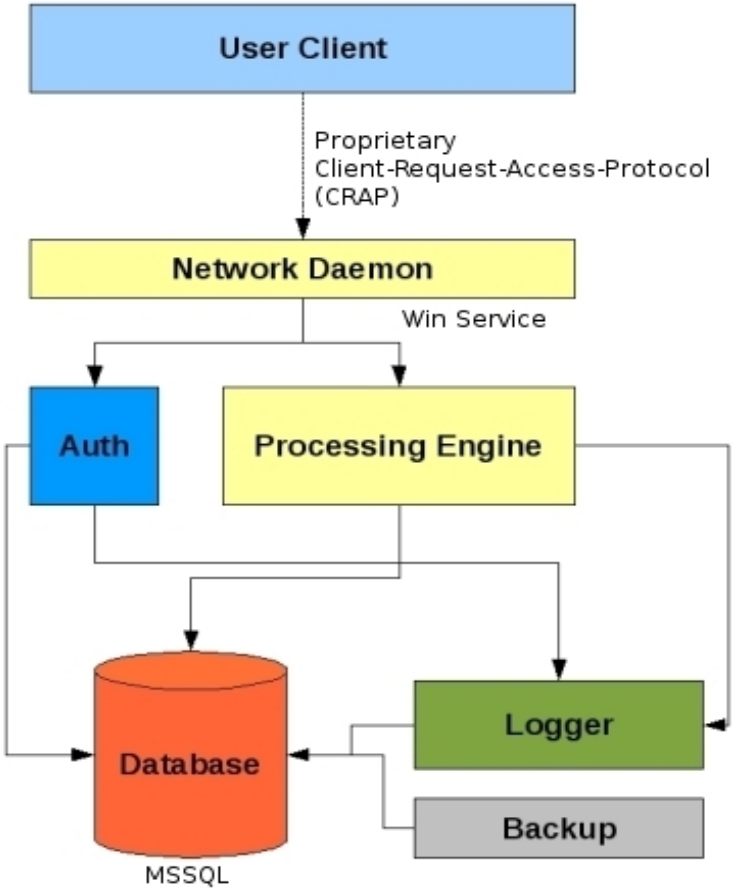
Methodology – Step 1: Scope

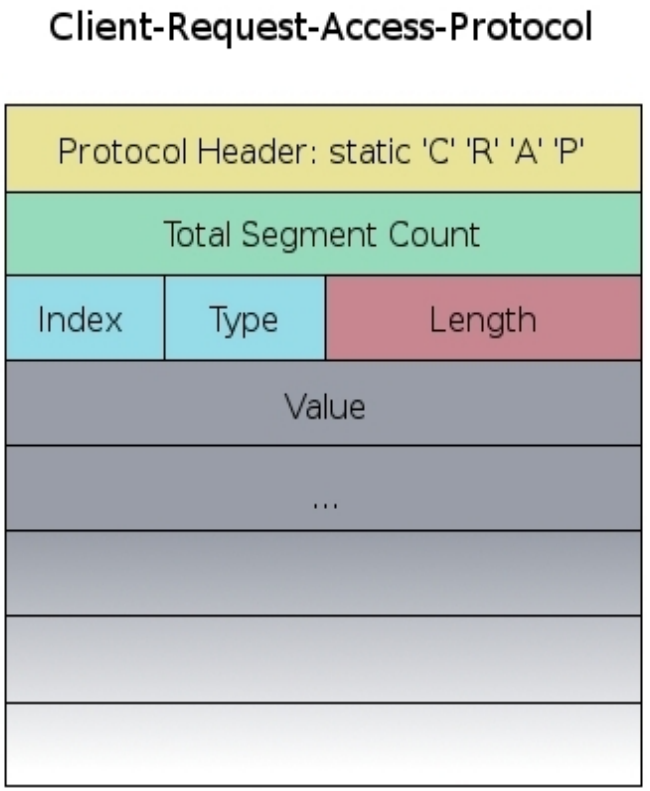
- Know the scope of the audit
  - How much code is there to read?
  - How much time is there to read it?
  - What is the purpose of the audit?
  - What tools are available to help?
  - What components are out-of-scope (missing, etc)?
- The hardest part is learning how to gauge time

Methodology – Step 2: Background

- Gather background information on the application
  - As much as possible
  - **Docs + Architecture Diagrams**
  - Talk with developers if possible
- Existing security concerns?
  - Developers usually have security skeletons in their closet
  - Often developers just don't have the time to fix them; security review results justify the time to management
  - Known problems are still problems. Don't let them gather dust.

Methodology – Example Arch Diagrams





Methodology – Step 3: Architecture + Purpose

- Knowing the purpose of the application(s)
  - Intended functionality – what is it *supposed* to do?
  - Users – who is supposed to use it? How?
  - Inputs – network / file / etc
  - Outputs – network / file / etc
  - Intended restrictions – what is it *not supposed* to do?
  - Other resources – what else does it interact with?

Methodology – Step 4: Big Picture Threats

- Think about high-level threats to the design
- Think outside of the realm of what the program was meant to do
  - In the intended design, what could introduce a threat to the system, users, etc?
  - What are the security implications of how the application achieves its goals?
  - How can the functionality be leveraged to an attackers advantage?
  - What type of security mechanisms were missed in the design?
  - Finally, skip what the program is, or meant to be, and instead consider what it can be used for. Game the system

Methodology – Step 4: Continued

- Keep this mind-set persistent throughout the audit
- These types flaws are missed by automated analysis tools!!! Architectural understanding is crucial!
- Examples:
  - Authentication / Authorization bypasses
  - Sensitive information disclosure
  - Cryptography use failures

Methodology – Step 5: Target Components

- Review and understand architecture
- Determine high-value code targets
- Focus on these targets
- Examples:
  - Network input/output code
  - Protocol and format parsing code
  - Database interaction; query generation

Methodology – Step 6: Component Exam

- Examine the logic in the targeted code area
  - Is the logic behind the design of each component sound?
  - Is the intended logic actually the logic that is implemented?
- Examine the component code implementation
  - Are there implementation bugs?
  - ...specifically in this case (C/C++), memory corruption?
  - Does user-influenced data touch resources?
  - ... if so, what resources and how? Implications?

Methodology – Step 7: Assess Risk

- Review the findings
- Likelihood
  - How likely is each vulnerability to be discovered?
  - How likely is each vulnerability to be successfully exploited
- Impact
  - What type of impact can the vulnerability have if exploited?
  - Is there anything that mitigates its impact?
- **Likelihood * Impact = Severity**

Methodology – Overview

- Understand full target purpose and architecture
- Determine which code targets are high-value
- Review high-value components
- Use product understanding along with potential likelihood and impact to determine severity of vulnerabilities
- Report!

Methodology – Tips

- Knowing when to give up on a bug
  - Tracking vulnerabilities back to their source can be painful
  - Often it is a far better use of time to make note of a time consuming bug and move on
  - If time permits, return to the bug and try and assess if its exploitable
  - Regardless, report the bug, even if only as an informational finding for the developers to look into further

Methodology – Tips

- Try and get a building copy of the target application code
  - All the code = all the context (at least, most of the context)
  - The compiler warnings are your friend, especially for tracking down certain data type bugs (discussed later)
  - Compiled binaries let you confirm and prove the existence of bugs

Memory Corruption – Introduction

- “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
- “crash” and “strange and bizarre program behavior” are developer code words for “...code is busticated, and can be exploited”

Memory Corruption – Implications

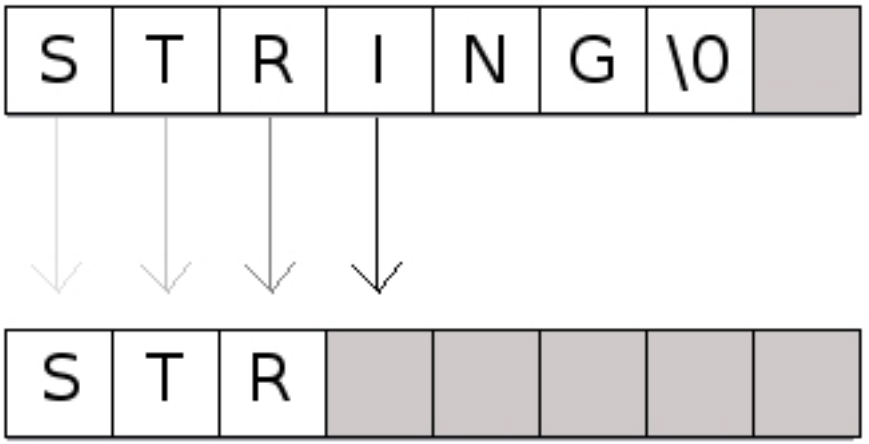
- Corruption of memory in the program could allow:
  - Compromise of stack frames, used by the program to maintain state of execution (among other things)
  - Compromise of meta-data, such as that used by the heap
  - Compromise of variables used by the program – think about that for a second
- This ultimately means memory corruption could allow:
  - At the very least, crashing the application :(
  - Altering logic of the program, by altering it's variables
  - Arbitrary code execution, by leveraging control program logic or internals (such as the stack frame)

Memory Corruption – Basic Example

- First example uses strcpy()
- Based on NULL-terminated C strings
- This function copies byte-by-byte from a source buffer to a destination buffer until a NULL byte is encountered in the source
- The NULL is then stored in the destination buffer to terminate the destination string

Memory Corruption – Basic Example

- char *strcpy(char *dest, const char *src);



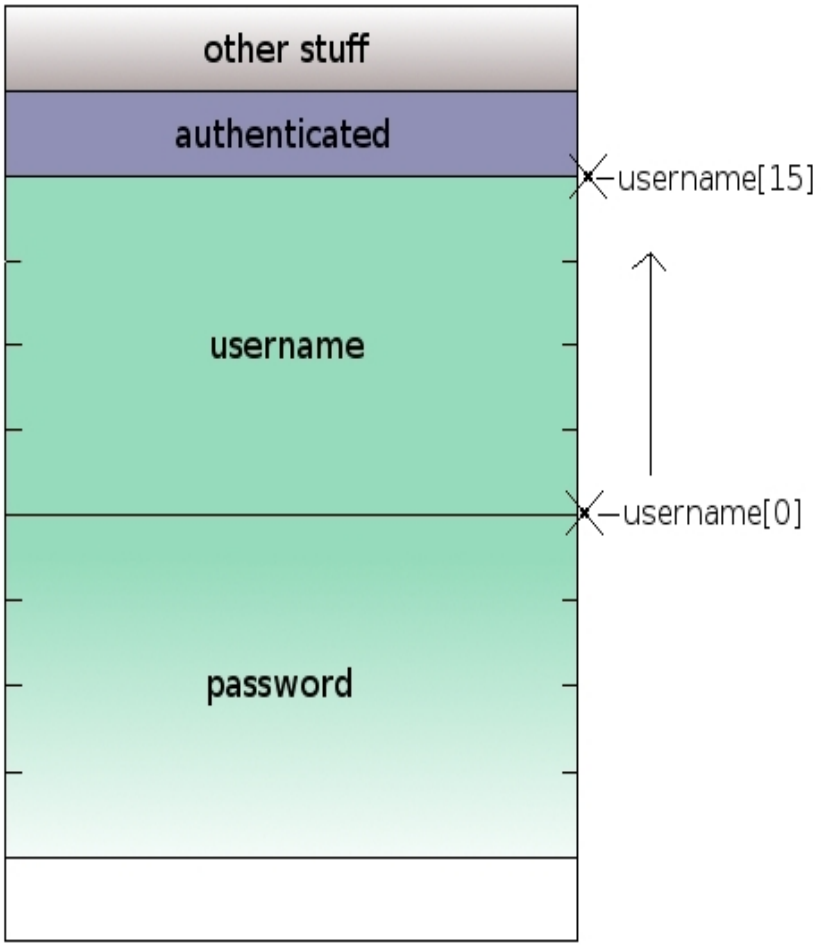
Memory Corruption – Basic Example

```
int login(const char *user, const char *pass)
{
    int authenticated = 0;
    char username[16];
    char password[16];


    strcpy (username, user);
    strcpy (password, pass);

    if (passwordLookup(&username, &password))
    {
        authenticated = 1;
    }

    return authenticated;
}
```

Memory Corruption – Basic Example Impact

- Here it is assumed the user is able to submit input
- The **strcpy()** will copy byte-for-byte into local stack array **username** until a NULL is encountered in **user**
- The **authenticated** integer (used as a sort of flag) sits higher on the stack
- The byte-for-byte copy done by **strcpy()** is unaware of any bounds to the buffer – if **username** is longer than 16 bytes, **strcpy()** will continue writing up the stack and into **autenticated**

Memory Corruption – Basic Example Visualized



Memory Corruption – Example Overview

- By supplying more than 16 bytes of non-NULL data for the user name input, the **strcpy()** will write past the buffer
- Writing past the buffer allows the user to write into the authentication variable, thus controlling it
- This alters the logic of the application; the user can be “authenticated” even if **passwordLookup()** does not authenticate them

Memory Corruption – Example Impact

- It gets worse
  - The program internally stores state information on the stack
  - The stack frame contains a stored location for execution, to execute after the function returns, called the EIP
  - Overwriting past **Authenticated** writes into the stack frame
  - This allows an attacker to control what code the program executes
  - This is how modern software exploitation was started
- Exploitation segment of the class will go into more detail and do hands-on exploiting

Memory Corruption – Example Impact

- Overwriting the stack frame is just one way to get code execution
- Heap meta-data, function pointers, many other useful targets to leverage
- Clever techniques innovated to leverage simple logic manipulation into code execution
- For now, just trust that in many (most) cases, memory corruption can be exploited in one form or another

Memory Corruption – Mind Set

- Observe the use of memory and how it is populated with data
- Try and visualize the sizes allocated and the data used
- Walk through loops and function use, considering the lengths of data used and the size of memory available
- Graph paper and pencil become useful with intricate code

Memory Corruption – Byte Sequence Overflows

- Byte-sequence overflows result from byte-for-byte manipulation: copies and moves
- Historically, the APIs used for string manipulation introduce most of the vulnerabilities of this nature
  - char *strcpy(char *dest, const char *src);
  - char *strcat(char *dest, const char *src);
  - int sprintf(char *str, const char *format, ...);
  - Lists full of tons of these APIs
- Many of these functions do not let the developer specify the amount of data to write

Memory Corruption – Byte Sequence Considerations

- Things to consider while auditing use of these APIs:
  - Where is the source of the data?
  - Where is the destination?
  - Are there size checks performed prior to use?
  - Is the source ever ensured to be NULL terminated?
  - Is the destination buffer dynamically allocated? How?
- Where in the code does this happen?
  - The closer to remotely controllable data, the worse (or better :P)
  - What is the purpose?

Memory Corruption – Byte Sequence Example 2

```c
#include <stdio.h>
#include <string.h>
#define STRING1 "a string"
 int main(int argc, char *argv[])
 {
     char buf1[32];
     char buf2[128];

     if (!argv[1])
         exit(1);

    strcpy(buf1, STRING1);
    printf("we're going to build a string!");
    sprintf(buf2, "%s: %s", buf1, argv[1]);
    printf("here it is: %s", buf2);
 }
```

Memory Corruption – Length Delineated

- String functions
  - char *strncpy(char *dest, const char *src, size_t n);
  - char *strncat(char *dest, const char *src, size_t n);
  - int snprintf(char *str, size_t size, const char *format, ...)
  - Many more..
- Input/Output functions
  - ssize_t read(int fd, void *buf, size_t count);
  - size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
  - ssize_t recv(int s, void *buf, size_t len, int flags);
  - etc.

Memory Corruption – Length Delineated

- Although these allow the restriction of the length of data...
  - Often developers use them incorrectly
  - These functions aren't perfect, even with the best intention
  - Just because developers can use them doesn't mean they choose to

Memory Corruption – Length Abuse

```
strncpy(buffer, userdata, strlen(userdata));

memcpy(buffer, userdata, strlen(userdata));
```

```
read(file, (char *)&usersize, 4);

read(file, buffer, usersize);
```

Memory Corruption – Length Abuse

- Consider string *concatenation*
  - Adding to a buffer which may have existing data
  - More data present means less space available for use
  - This gets very dangerous, especially in loops

```
strncat(buf, userdata, sizeof(buffer)-1);
```

Should really be:

```
strncat(buf, userdata,
          sizeof(buffer) - strlen(buffer)-1);
```

Memory Corruption – Length Abuse

- Confusion between **size** and **count**
  - Not all counts are measured in byte size
  - Functions using multi-byte characters often measure in count of characters
  - Conversions, copies, etc
- Example functions

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

int MultiByteToWideChar(UINT CodePage, DWORD dwFlags,
          LPCSTR lpMultiByteStr, int cbMultiByte,
          LPWSTR lpWideCharStr, int cchWideChar);
```

Memory Corruption – Length Abuse

- These conversion functions measure the destination length in **wide characters**
- “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*
- sizeof( ) only determines the size of **bytes**...

Memory Corruption – Length Abuse

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

void funcXYZ(int socket)
{
    wchar_t wcbuf[100];
    char *userdata;
    userdata  = ImaginaryGetDataFromNetwork(socket);
    mbstowcs(wcbuf, userdata, sizeof(wcbuf));
    processData(wcbuf);
    return;
}
```

Memory Corruption – Pointer Byte Sequences

- Developers often perform similar tasks with pointers

```
...
  char *p1 = buf;
  char *p2 = userdata;
  while (*p2)
  {
      if (*p2 == 'x')
      {
          break;
      }
      *p1++ = *p2++;
  }
...
```

Memory Corruption – Pointer Byte Sequences

- Tedious to read
- Often fraught with bugs
- Common in parsing code: file formats, protocols, etc
- Common in home-grown encoding/conversion functions
- Sometimes a length is specified in the loop, but determined by ***the source data***
- Examine what initiates and concludes the loops, then busticate accordingly