Vulnerability Analysis: Source Code Auditing Part I

Brandon Edwards

drraid gmail com

Why Am I Here?

- I enjoy finding vulnerabilities in code
- I work at a security company where a big part of my job is source code auditing
- I have read millions of lines of code. seriously.
- I have a jar on my desk full of developer souls

Goals

- My goal is to make you a ninja at code auditing
  - Know what vulnerabilities look like in code
  - Find critical, devastating vulnerabilities
  - Approach and grasp large code bases

Source Code Auditing

- Day 1:
  - What is source code auditing
  - Methodology
  - Approaches to reading code
  - Tools
  - Introduction to bug classes

What is it?

- The process of reading source code in order to find vulnerabilities
- Very effective at identifying implementation security problems
  - Memory corruption
  - Resource influence
  - Race conditions
  - In-line data injection
- Compliments architectural review

Methodology

- Employ and strictly adhere to a methodology
- Example structure for methodology:
  - Scope
  - Understand architecture, application purpose
  - Identify attack surface, target components
  - Set objectives for concerns & coverage
  - Allot reasonable time per component, per objective
  - Review, execute on strict schedule

Methodology - Scope

- Scope of the code review can be broken down in to three major points
  - Purpose of the review
  - Amount of code to review
  - Amount of time available

Methodology Scope: Purpose

- There can be different reasons to be auditing code
  - Client specifically requested code audit
    - To find the most exploitable bugs
    - To find the most potential bugs
  - Penetration testing
    - Client is running open source product
    - Access to source is attained
      - Source disclosure
      - Stolen from dev servers
      - Floating in the underground

Methodology Scope: Amount

- How much code is there to read
  - What is the size of the code base
  - How many 3rd party components are there
  - How dense is the code
  - What language is it?
  - ...And how many languages interact?

Methodology Scope: Time

- There is never enough time
- The most difficult part of the audit is gaging how long it will take
  - To read and fully understand the code base
  - To track a bug back & determine exploitability
- Time is why a solid methodology with key targets is crucial

Methodology - Architecture

- Gain an architectural understanding
- Architectural + design documentation is very useful
- Talk with developers when possible
- Results from threat modeling / architectural review are even more useful
- *Part of source review is ensuring the application actually works as designed*

Methodology – Targeting

- Reflect on Architecture to identify targets
  - Sources of input
    - Remote input
    - Influence on the application
  - Security mechanisms
  - Complex data handling, parsing
  - State management

Methodology - Objectives

- Set objectives based on identified targets
- Base objectives on desired coverage and concerns:
  - Verify proper implementation of security mechanisms, proper use of crypto
  - Determine what type of vulnerabilities each target might be prone to
  - Set goals on vulnerability types to look for, key points to hit on while reviewing the code

Methodology - Objectives

- Objectives set based on concerns are intended to give focus to guide the review; do not let them rule out other possible attacks
- Prioritize the objectives based on severity
  - Likelihood of vulnerability, or ease of being reached
  - Impact of exploitation

Methodology – Execution

- Allot and manage time for each established objective
- Execute based on a schedule, engaging the top of the objective priority list first, working down
- Do not go over time for each component
  - Sometimes bugs are hard to track back, leave it alone & come back if time permits
  - If all else fails tag the bug as “possibly exploitable”, “problematic”

Reading Code

- Code base will probably be HUGE
- Understanding the code base can be difficult
- Read the code iteratively
  - Initial reading can be frustrating
  - Realize that components will not make sense without context
  - It may be necessary to return to a previously skipped piece of code

Reading Code

- Various types of tools exist to help audit code
  - Editors
  - Parsers / pattern matching
  - Static analyzers
  - Pen and pad ;)

Reading Code: Tools

- Free syntax aware editors
  - VI / VIM
  - Emacs
  - Gedit
  - Source Navigator
  - Eclipse
- Commercial Products
  - Visual Studio
  - Source Insight

Reading Code: Tools

- Commercial static analysis tools
  - Klocwork
  - Fortify
  - Coverity
- Free tools
  - RATS
  - FindBugs (Java)

Reading Code: Tools

- Static analysis tools are OK
  - Often miss vulnerabilities
  - Have many false positives
- They can also help in just understanding a code base
- Ultimately, nothing will replace a sharp minded, living, breathing auditor

Reading Code: Tools

- Source Navigator Demo
- Quick hack tips:
  - FIXME, XXX, WARNING
  - Look for developer bad habits, patterns
  - Date/Time stamp files if possible
    - Really old, untouched code
    - Code checked in late at night
    - Last minute commits
  - Novice developers
  - Reimplementation in other products

Implementation Bugs

- A bug in how something is implemented! (LOL SURPRISE!)
- From a security standpoint, these really matter if they are:
  - In implementation of a security mechanism
  - Have effects in application state or memory

Implementation Bugs

- Implementation bugs are logical faults
- They can result from many conditions, some are:
  - Trusting or assuming user/remote input
  - Misuse or misunderstanding of APIs
  - Miscalculation of an operation
  - Failure to verify the result of an operation
  - Failure to ensure & keep track of state

Implementation Bugs

- Notorious for occurring in complex code
- Parsing and state management
  - File format
  - Network protocol
  - Relational data management

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 and bizarre program behavoir”

  is just fancy developer speak for:
  “Busticated, exploitable and pwned”
- There is no such thing a crash or DoS, there are only vulnerabilities that *you* cannot exploit

  – Ryan Permeh

Memory Corruption

- Basic implications, consider stack-based buffer overflow:

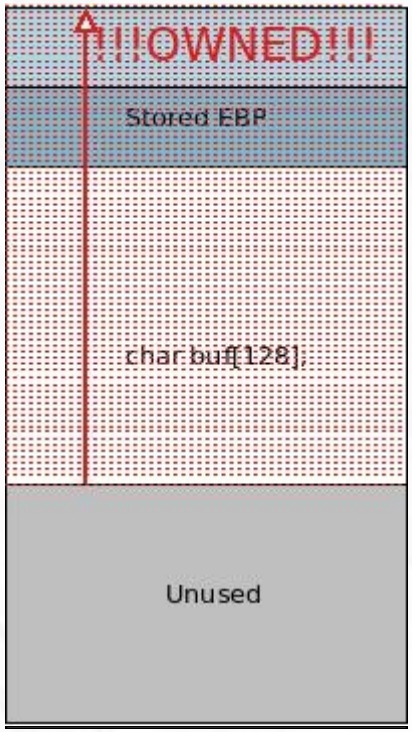
```
int vuln_function(char *userstring)
{
    char buf[128];
  /* first make local copy of data to manipulate */
    strcpy(buf, userstring);

    /* ... */
    return;
}
```

Memory Corruption

- The strcpy() function performs unbounded copy of data
- In this example, this is onto the function's local stack
- This means if an attacker could reach this, they could control contents on the stack

Memory Corruption



Memory Corruption

- Unbounded byte-by-byte copies
- Copy data a byte at a time from a source to a destination, until a delimiter in the source is hit
- Provides no direct means of restricting or knowing how much data will be copied
- Lots of these APIs
- Responsible for many many vulnerabilities historically

Memory Corruption

- Unbounded byte-by-byte copy/read API examples
  - strcpy()
  - strcat()
  - sprint()
  - gets()
  - sscanf()
  - etc..
- These APIs do not take a maximum length parameter

Memory Corruption

- Also often seen in pointer arithmetic:

```
int vuln_function(char *userstring)
{
    char buf[128];
    char *src, *dst;

    src = userstring;
    dst = buf;

    while(*src != 0x0)
    {
        *dst++ = *src++;
    }

    /* ... */
    return;
}
```

Memory Corruption

- Functions which perform unbounded data manipulation are obviously bad
- Equivalent bounded API functions exist...

Memory Corruption

- Bounded byte copy APIs
  - strncpy()
  - snprintf()
  - MultiByteToWideChar(), mbstowcs()
  - strncat()
- These are even more fun to own, and lead to other dangerous scenarios as developers misuse them...

Memory Corruption

- Bounded string copy example:

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

....Now there's a parameter to enforce length...

Memory Corruption

...The problem here is that the size parameter is very often abused..

```
int vuln_function(char *userstring)
{
    char buf[128];

    strncpy(buf, userstring, strlen(userstring));

    /* ... */
    return;
}
```

Memory Corruption

- Copy Pasta Abuse!

```
int vuln_function(char *string)
{
    char buf1[256];
    char buf2[256];
    char buf3[128];

    strncpy(buf1, string, sizeof(buf1));
    strncpy(buf2, string, sizeof(buf1));

    /* ... */
    strncpy(buf3, string, sizeof(buf1));

    return;
}
```

Memory Corruption

- Also, these what this size **actually** means is not always clear...

```
int vuln_function(char *name, char *password)
{
    char buf1[128];
    char buf2[128];
    char buf3[128];

    strncpy(buf2, name, sizeof(buf2));
    strncpy(buf3, password, sizeof(buf3));

    /* Welp, we already know the length, this is safe! */
    strcpy(buf1, buf3);

    return; }
```

Memory Corruption

- Okay.. so now it's clear, right?

  strncpy(buf, string, sizeof(buf)-1);

  ...Fine.. that's safe.. what about other functions?

Memory Corruption

- strncat() - string concatenation

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

- Max length of the copy can be specified..
- But, this is still commonly misused

Memory Corruption

- Consider this scenario..

```
int vuln_function(char *string1, char *string2)
{
    char buf1[256];

    strncpy(buf1, string1, 50);
    strncat(buf1, string2, sizeof(buf1)-1);

    return;
}
```

Memory Corruption

- Properly done:

  strncat(buf, string, sizeof(buf) - strlen(buf) -1);

- **NOTICE, strncat() can write up to the length specified +1 NULL byte:**

  strncat(buf, string, sizeof(buf) – strlen(buf));

- **The above will result in a out of bounds write of 1 NULL byte beyond the end of buf**