

THE SCIENCE OF CODE AUDITING

Mark Dowd

Neel Mehta

Alex Wheeler

SUMMARY

1. Introduction
2. Code Survey – What to Audit
3. Methodology – How to Audit
4. Source & Binary Parallels
5. Questions

INTRODUCTION

Informal Definition:

- Structured manual review of code to identify security vulnerabilities
- Primary efforts are focused on static analysis
- Runtime analysis is relied upon primarily for verification purposes

INTRODUCTION

Toolset:

- IDA is the best tool available for binary static analysis
- ctags & cscope, sourcenv are good for source code
- SoftIce/OllyDbg on Microsoft and gdb on others for runtime analysis/verification
- Vmware useful for testing vulnerabilities on different target versions

INTRODUCTION

Code Auditing Success Factors:

- API, OS, and machine background knowledge
- Pattern recognition
- Application understanding
- Leave no code unaudited

INTRODUCTION

Background Knowledge:

- The more familiar you are with the machine, OS, and API's, the more successful audits will be too
- API, OS, and machine quirks and pitfalls (we will see some of these)
- External entities, special handling (/dev files, named pipes, etc.), signals/events, etc.

INTRODUCTION

Pattern Recognition:

- Code constructs
- Dangerous use of API's
- Flawed logic

INTRODUCTION

Functional Understanding:

- Complements pattern recognition
- Identifying where code can be influenced
- Utilization of available documentation (RFC's, protocol specs, product-specific docs)

INTRODUCTION

Completeness:

- Thoroughness is important because the vast majority of code is usually ok
- When you make assumptions about how something works, you either miss bugs or assume something is a bug when it is not

CODE SURVEY

It is impossible to cover all interesting code in a speech, but here are some big hitters.

- API Based Bugs
- External Resource Interactions
- Programming Construct Errors
- State Mechanics

CODE SURVEY

API Based Bugs – based on misuse of API's provided by the OS or application.

- Dangerous string or formatting functions: e.g., `sprintf()`, `strcpy()`, `strcat()`, `printf()`, `syslog()`...
- Dangerous implicit behavior: e.g., Allocators that round
- Cumbersome/Complicated API reference contents: e.g., threading, IPC

CODE SURVEY

API Based Bug Example 1:

```
char blah[260], buf[256];  
sprintf(blah, "%s", "BLAH");  
recv(socket, buf, 256, 0);  
strncat(blah, buf, 256);
```

CODE SURVEY

API Based Bug Example 2:

```
int allocator(struct memory *h, int length){
    while(h->next != 0)
        h = h->next;

    h->next = calloc(length + 4, 1);

    return h->next + 4;
}
```

CODE SURVEY

External Resource Interactions – bugs where the application interacts dangerously with other entities.

- Privilege escalation through RPC/COM/Pipes and other forms of IPC
- Executing external programs via `system()` - metacharacters
- Executing external programs via `execve()/CreateProcess()` - polluting the environment, fd leaks, etc.
- File interaction: doubledots, special files (`/dev/`, `LPT0`, `ADS's`, etc.)

CODE SURVEY

External Resource Interactions Example 1:

```
HANDLE GetRequestedFile(LPCSTR requestedFile)
{
    if(strstr(requestedFile, ".."))
        return INVALID_HANDLE_VALUE;

    if(strcmp(requestedFile, ".config") == 0)
        return INVALID_HANDLE_VALUE;

    return CreateFile(requestedFile, GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
}
```

CODE SURVEY

External Resource Interactions Example 2:

```
char *ProfileDirectory = "c:\profiles";
```

```
BOOL LoadProfile(LPCSTR UserName) {  
    HANDLE hFile; char buf[MAX_PATH];
```

```
    if(strlen(UserName) > MAX_PATH - strlen(ProfileDirectory) - 12) return FALSE;
```

```
    snprintf(buf, sizeof(buf), "%s\prof_%s.txt", ProfileDirectory, UserName);
```

```
    hFile = CreateFile(buf, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
```

```
    if(hFile == INVALID_HANDLE_VALUE) return FALSE;
```

```
    // ... load profile data ...
```

```
}
```

CODE SURVEY

Programming Construct Errors – the bugs are the result of bad programming constructs.

- Integer signedness
- Integer boundaries
- Checks that are logically wrong or susceptible to integer problems
- Loops that have bad boundaries
- Unchecked variables

CODE SURVEY

Programming Construct Error Example 1:

```
static int CAB_read_record(CAB_FILE__struct *cfs, BYTE *dst) {
    BYTE tmp = 0;
    int count = 0;

    do {
        count++;
        cfs->CAB_fgetc(cfs, &tmp);
        if(dst) {
            *dst++ = tmp;
        }
    } while(tmp);
    ...
    Return count;
}
```

CODE SURVEY

Programming Construct Error Examples 2 & 3:

```
#define MAXSTRLEN 100
...
char tmp[256];
char smallbuf[MAXSTRLEN+1];

recv(socket, tmp, 256, 0));

if(MAXSTRLEN < 1 + tmp[0])
    memcpy(smallbuf, tmp+1, MAXSTRLEN);
else
    memcpy(smallbuf, tmp+1, tmp[0]);
```

CODE SURVEY

Programming Construct Error Example 4:

```
▶ LOOP:
  mov edx, [esi+198]      ;current offset into large output buffer
  mov ecx, [esi+190]     ;ptr to start of small user controlled data
  dec edx
  mov [esi+198], edx
  mov eax, edx
  mov edx, [esi+1A0]     ;current index
  mov cl, [ecx, edx]
  mov [eax], cl
  mov edx, [esi+1A0]     ;current index
  mov eax, [esi+18C]     ;small un-trusted table
  mov eax, [eax+edx*4]
  cmp eax, FF
  mov [esi+1A0], eax    ;current index
  ja LOOP
```

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()
```

CODE SURVEY

State Mechanics – these bugs are where the program is left in an inconsistent state.

- Thread safety issues
- Async-safety issues (signals)
- Global variables left in an undesired state

CODE SURVEY

State Mechanics Bug Example 1:

```
From buffer_append_space(): // buffer is global
    buffer->alloc += len + 32768;
    if (buffer->alloc > 0xa00000)
        fatal("buffer_append_space: alloc %u not
supported", buffer->alloc);
    buffer->buf = xrealloc(buffer->buf, buffer->alloc);
    goto restart;

/* Frees any memory used for the buffer. */
void buffer_free(Buffer *buffer) {
    memset(buffer->buf, 0, buffer->alloc);
    xfree(buffer->buf);
}
```

CODE SURVEY

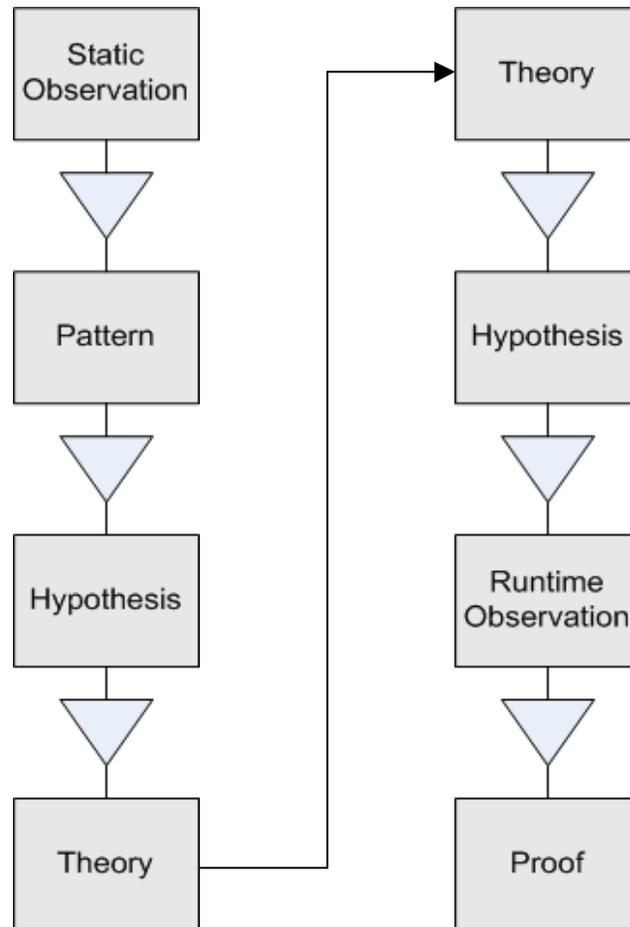
State Mechanics Bug Example 2:

```
// global
request *head
void server_thread() {
    while(1) {
        if(request_available()) {
            get_request(head);
            CreateThread(NULL,0,
                processing_thread_entrypoint,
                NULL,0);
        } else
            wait_for_request();
    }
}
```

```
void processing_thread_entrypoint() {
    request *req;
    // find first unprocessed request
    for(req=head;req && !req-
        >processed;req = req->next);
    if(req) {
        req->processed = 1;
        process_request(req);
    }
    ExitThread(0);
}
```

METHODOLOGY

Induction
(Hunt)



Deduction
(Verify)

METHODOLOGY

Inductive Process

- Hunt
 - annotating
 - following x-refs
 - reversing logic

Deductive Process

- Verify
 - after static analysis fails to reveal dizz, rely on runtime analysis for ultimate proof

METHODOLOGY

Hunt - Annotate Code:

- Annotation should occur in all phases, but is a necessary 1st step
- Input vectors
 - network
 - files
 - IPC
- Be mindful some vectors are indirect

METHODOLOGY

Hunt - Annotate Code Continued:

- Core input utility procedures
 - crc, checksum, etc.
 - byte ordering, data representations
 - context specific processing
- Memory routines
 - allocation and resizing
 - free
 - copy

METHODOLOGY

Hunt - Follow X-refs:

- Input vectors
- Utility procedures
- Memory procedures
- Dealing with external entities (creating processes, file manipulation, pipes/rpc, etc.)

METHODOLOGY

Hunt - Follow X-refs Continued:

- Continue annotating
 - wrapper functions
 - arguments
 - structures/classes
 - local variables
- Example 1.0

METHODOLOGY

Repeat:

– Induction

- use newly applied knowledge of global structures from other parts of the code
- allows analysis of input further from initialization, generate additional annotation, hypothesize or resolve indirection
- aids recognition of context specific processing (e.g., file formats, network protocols, processing algorithms)

– Example 1.1

METHODOLOGY

Verify:

- Statically backtrace to eliminate false bugs and identify the vulnerability context
 - continue to annotate code
 - tracing into code past potential bugs is also valuable
- Generate normal event to trigger code
 - aids in resolving/verifying indirection
 - if trigger fails systematically move break point back in the call tree to reveal reason
 - getting dizzed in this step motivates you to do more thorough static analysis next time

METHODOLOGY

Verify Continued:

- Generate vulnerability event to trigger code
 - usually best to do this w/ minimal effort
 - same as before - if trigger fails systematically move break point back in the call tree
 - getting dizzed here is sometimes unavoidable ☹
- Example 1.2

SOURCE & BINARY PARALLELS

Source Code Advantages:

- Annotation
 - developer notes, application knowledge
 - very little time spent here, relative to binary audits
- Abstraction – high level logic is more apparent
- Locating version differences is trivial (although SABRE Bindiff usually eliminates this advantage)

SOURCE & BINARY PARALLELS

Source Code Challenges:

- Some bugs are more subtle in source form
 - machine specifics are only implied, e.g., sign extensions and conversions
- Developers' annotation carries implicit meaning, which can be misleading
- If source code is public, often you need to find subtle vulnerabilities

SOURCE & BINARY PARALLELS

Binary Code Advantages:

- You do all the code annotation, which can be more powerful than developer annotation
- It is possible this code has been reviewed to a lesser extent

SOURCE & BINARY PARALLELS

Binary Code Challenges:

- Binary audits require more time than source
 - annotation, reversing program logic
 - potentially need to overcome obfuscation (either deliberately obfuscated code or code that is difficult to understand due to compiler optimization)
- Indirection can be annoying to resolve statically
- High-level design vulnerabilities can be hard to understand

SOURCE & BINARY PARALLELS

Really no difference in basic methodology

- Binary generally requires more time

Interpreting binary as source

- Compiler-specific constructs
- Machine-specific constructs
- Annotation
- Indirection

Thank You

Questions?