Vulnerability Extrapolation
USENIX WOOT 2011

Fabian ‘fabs’ Yamaguchi
Recurity Labs GmbH, Germany
Agenda

- Patterns you find when auditing code
- Exploiting these patterns: **Vulnerability Extrapolation**
- Using machine learning to get there
- A method to assist in manual code audits based on this idea
- The method in practice
- A showcase
Exploring a new code base

- Like an area of mathematics you don’t yet know.
- It’s not completely different from the mathematics you already know.
- But there are secrets specific to this area:
  - Vocabulary
  - Reoccurring patterns in argumentation
  - Weird tricks used in proofs
- Understanding the specifics of the area makes it a lot easier to reason about it.
static int
TIFFFetchShortPair(TIFF* tif, TIFFDirEntry* dir)
{
    switch (dir->tdir_type) {
        case TIFF_BYTE:
            case TIFF_SBYTE:
            {
                uint8 v[4];
                return TIFFFetchByteArray(tif, dir, v)
                    && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
            }
        case TIFF_SHORT:
        case TIFF_SSHORT:
        {
            uint16 v[2];
            return TIFFFetchShortArray(tif, dir, v)
                && TIFFSetField(tif, dir->tdir_tag, v[0], v[1]);
        }
        default:
            return 0;
    }
}
Another Example: libTIFF
CVE-2006-3459 | CVE-2010-2067

```c
# Another implementation...

static int
TIFFFetchSubjectDistance(TIFF* tif, TIFFDirEntry* dir)
{
    uint32 l[2];
    float v;
    int ok = 0;

    if (TIFFFetchData(tif, dir, (char*)l) &
        cvtRational(tif, dir, l[0], l[1], &v)) {
        /*
         * XXX: Numerator 0xFFFFFFFF means that we have infinite
         * distance. Indicate that with a negative floating point
         * SubjectDistance value.
         */
        ok = TIFFSetField(tif, dir->tdir_tag,
                          (l[0] != 0xFFFFFFFF) ? v : -v);
    }

    return ok;
}
```
LibTIFF: Bug Analysis

- TIFFFetchShortArray is actually a wrapper around TIFFFetchData.
- The two are pretty much synonyms.
- These functions are part of an API local to libTIFF.
- Badly designed API: the amount of data to be copied into the buffer is passed in one of the fields of the dir-structure and not explicitly!
- Developers missed this in both cases and it’s hard to blame them.
The times of “grep ‘memcpy’ /*.c” may be over. But that does not mean patterns of API use that lead to vulnerabilities no longer exist!
Vulnerability Extrapolation

- Given a function known to be vulnerable, determine functions similar to this one in terms of application-specific API usage patterns.
- Vulnerability Extrapolation exploits the information leak you get every time a vulnerability is disclosed!
What needs to be done

- We need to be able to determine how “similar” functions are in terms of dominant programming patterns.
- We need to find a way to extract these programming patterns from a code-base in the first place.
- How do we do that?
Similarity – A decomposition

Signal Processing: Decomposition into components of different frequencies: Noise is suspected to be of high frequency while the signal is of lower frequency.

Decomposition into shape and rotation: If rotation is just a detail, these are pretty similar.

In Face-Recognition, faces are decomposed into weighted sums of commonly found patterns + a noise-term.
Think of it as ‘zooming out’

\[ \approx x_1 + x_2 + x_3 \]

Increasing level of detail/frequency

\[ \approx x_1 + x_2 + x_3 \]

Decreasing dominance of pattern

Linear approximation of each function by the most dominant API usage patterns of the code-base it is contained in!
Extracting dominant patterns

How do we identify the most dominant API usage patterns of a code-base?

In Face Recognition, a standard technique is Principal Component Analysis.
Mapping code to the vector space

- Describe functions by the API-symbols they contain.
- API-symbols are extracted using a fuzzy parser.
- Each API-symbol is associated with a dimension.

```c
func1()
{
    int *ptr = malloc(64);
    fetchArray(pb, ptr);
}
```

```
malloc  (1)
printf (0)
int     1
::
fetchArray (1)
```
**Principal Component Analysis**

Data Matrix (Contains all function-vectors)

Each column of $U$ is a dominant pattern.

Each row is a representation of an API-symbol in terms of the most dominant patterns.

**Strength of pattern**

$$M \approx U \Sigma V^T =
\begin{pmatrix}
\leftarrow u_1 \rightarrow \\
\leftarrow u_2 \rightarrow \\
\vdots \\
\leftarrow u_{|S|} \rightarrow \\
\end{pmatrix}
\begin{pmatrix}
\sigma_1 & 0 & \ldots & 0 \\
0 & \sigma_2 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & \sigma_d \\
\end{pmatrix}
\begin{pmatrix}
\leftarrow v_1 \rightarrow \\
\leftarrow v_2 \rightarrow \\
\vdots \\
\leftarrow v_{|\chi|} \rightarrow \\
\end{pmatrix}^T$$

Representation of functions in terms of the most dominant patterns
In summary

Code base of functions

(1) Extraction of API symbols

(2) Embedding in vector space

(3) Identification of API usage patterns

(4) Assisted vulnerability discovery
void guiFunc1(GtkWidget *widget)
{
    int j;
    gui_make_window(widget);
    GtkWidget *button;
    button = gui_new_button();
    gui_show_window();
}

void guiFunc2(GtkWidget *widget)
{
    gui_make_window(widget);
    GtkWidget *myButton;
    button1 = gui_new_button();
    button2 = gui_new_button();
    button3 = gui_new_button();

    for(int i = 10; i != i; i++)
        do_gui_stuff();
}
void netFunc1()
{
    int fd;
    int i = 0;
    struct sockaddr_in in;
    fd = socket(arguments);
    recv(fd, moreArguments);

    if(condition){
        i++;
        send(fd, i, arg);
    }
    send(fd, i, arg);
    close(fd);
}

void netFunc2()
{
    int fd;
    struct sockaddr_in in;
    hostent host;
    fd = socket(arguments);
    recv(fd, moreArguments);
    gethostbyname(host)

    if(condition){
        int i = 0;
        i++;
        send(fd, i, arg);
    }
    send(fd, i, arg);
    close(fd);
}
void listFunc1(int elem)
{
    GList myList;
    if(! list_check(myList)){
        do_list_error_stuff();
        return;
    }
    list_add(myList, elem);
}

void listFunc2(int elem)
{
    GList myList;
    if(! list_check(myList)){
        do_list_error_stuff();
        return;
    }
    list_remove(myList, elem);
    list_delete(myList);
}
Projection onto the first two principal components

Core API

Functions

Occurs in this context but does not constitute the pattern
Vulnerability Extrapolation

- Take a function that used to be vulnerable as an input.
- Measure distances to other functions to determine those functions, which are most similar.
- Let’s try that for FFmpeg.
Original bug: CVE-2010-3429

```c
static int flic_decode_frame_8BPP(AVCodecContext *avctx,
                                 void *data, int *data_size,
                                 const uint8_t *buf, int buf_size)
{
    pixels = s->frame.data[0];
    case FLI_DELTA:
        y_ptr = 0;
        compressed_lines = AV_RL16(&buf[stream_ptr]);
        stream_ptr += 2;
        while (compressed_lines > 0) {
            line_packets = AV_RL16(&buf[stream_ptr]);
            stream_ptr += 2;
            if (((line_packets & 0xC000) == 0xC000) {  // line skip opcode
                line_packets = -line_packets;
                y_ptr += line_packets * s->frame.linesize[0];
            } else if (((line_packets & 0xC000) == 0x4000) { ...
                ...
            } else if (((line_packets & 0xC000) == 0x8000) { // "last byte" opcode
                pixels[y_ptr + s->frame.linesize[0]-1] = line_packets & 0xff;
            } else { ...
                     ...
            }
            y_ptr += s->frame.linesize[0];
        }
    break;
    [...]}
```

**Decoder-Pattern:**

- Usually a variable of type `AvCodecContext`.
- AV_RL*-Functions used as sources.
- Lot's of primitive types with specified width used.
- Use of `memcpy`, `memset`, etc.

Unchecked index,
Write to arbitrary location in memory.
Extrapolation

- The closest match contained the same vulnerability but it was fixed when the initial function was fixed.

<table>
<thead>
<tr>
<th>Score 1</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000000</td>
<td>flic_decode_frame_8BPP (libavcodec/flicvideo.c)</td>
</tr>
<tr>
<td>0.964096</td>
<td>flic_decode_frame_15_16BPP (libavcodec/flicvideo.c)</td>
</tr>
<tr>
<td>0.826979</td>
<td>lz_unpack (libavcodec/vmdav.c)</td>
</tr>
<tr>
<td>0.803331</td>
<td>decode_frame (libavcodec/lcidec.c)</td>
</tr>
<tr>
<td>0.796700</td>
<td>raw_encode (libavcodec/rawenc.c)</td>
</tr>
<tr>
<td>0.756951</td>
<td>vmdvideo_decode_init (libavcodec/vmdav.c)</td>
</tr>
<tr>
<td>0.723750</td>
<td>vmd_decode (libavcodec/vmdav.c)</td>
</tr>
<tr>
<td>0.702356</td>
<td>aasc_decode_frame (libavcodec/aasc.c)</td>
</tr>
<tr>
<td>0.684610</td>
<td>flic_decode_init (libavcodec/flicvideo.c)</td>
</tr>
<tr>
<td>0.665167</td>
<td>decode_format80 (libavcodec/vqavideo.c)</td>
</tr>
<tr>
<td>0.664279</td>
<td>targa_decode_rle (libavcodec/targa.c)</td>
</tr>
<tr>
<td>0.660454</td>
<td>adpcm_decode_init (libavcodec/adpcm.c)</td>
</tr>
<tr>
<td>0.659811</td>
<td>decode_frame (libavcodec/zmbv.c)</td>
</tr>
<tr>
<td>0.655338</td>
<td>decode_frame (libavcodec/8bps.c)</td>
</tr>
<tr>
<td>0.651587</td>
<td>msrle_decode_8_16_24_32 (libavcodec/msrledec.c)</td>
</tr>
<tr>
<td>0.648321</td>
<td>wmavoice_decode_init (libavcodec/wmavoice.c)</td>
</tr>
<tr>
<td>0.646872</td>
<td>get_quant (libavcodec/nuv.c)</td>
</tr>
<tr>
<td>0.641871</td>
<td>MP3lame_encode_frame (libavcodec/libmp3lame.c)</td>
</tr>
<tr>
<td>0.641642</td>
<td>mpegts_write_section (libavformat/mpegtsenc.c)</td>
</tr>
<tr>
<td>0.634922</td>
<td>tgv_decode_frame (libavcodec/eatgv.c)</td>
</tr>
</tbody>
</table>
static void vmd_decode(VmdVideoContext *s) {
    [...]
    int frame_x, frame_y;
    int frame_width, frame_height;
    int dp_size;
    frame_x = AV_RL16(&s->buf[6]);
    frame_y = AV_RL16(&s->buf[8]);
    frame_width = AV_RL16(&s->buf[10]) - frame_x + 1;
    frame_height = AV_RL16(&s->buf[12]) - frame_y + 1;
    [...]
    if (s->size >= 0) {
        /* originally UnpackFrame in VAG's code */
        pb = p;
        meth = *pb++;
        [...]
        dp = &s->frame.data[0][frame_y * s->frame.linesize[0] + frame_x];
        dp_size = s->frame.linesize[0] * s->avctx->height;
        pp = &s->prev_frame.data[0][frame_y * s->prev_frame.linesize[0] + frame_x];
        switch (meth) {
            [...]
            case 2:
                for (i = 0; i < frame_height; i++) {
                    memcpy(dp, pb, frame_width);
                    pb += frame_width;
                    dp += s->frame.linesize[0];
                    pp += s->prev_frame.linesize[0];
                }
                break;
            [...]
        }
    }
}
Summary

- Often inherent link between vulnerabilities and API usage patterns
- Application of machine learning for automatic identification of these patterns
- Extrapolation of known vulnerabilities using dominant API usage patterns
- Discovery of a 0-day vulnerability in a widely used application
Questions?

Recurity Labs GmbH, Berlin, Germany
http://www.recurity-labs.com

Fabian Yamaguchi
Vulnerability Researcher
fabs@recurity-labs.com

Recurity Labs GmbH, Berlin, Germany
http://www.recurity-labs.com