CMSC389R

Forensics II

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

announcements

- How was Kaizen CTF?
- If you haven’t yet, pick up your midterms!
- Binaries II questions?

forensics I refresher

- Two stages: recovery/extraction and analysis
- Recovery tools:
  - Physical media: dd, EnCase, FTK
  - “Live” sources:
    - Network: Wireshark, tcpdump
    - Memory: /dev/{k,}mem, /proc/$pid/maps
- Analysis tools:
  - Content-agnostic: strings, binwalk, file
  - Content-aware: Wireshark (again)
  - Metadata: exiftool
  - Steganography: steghide, stegdetect

today’s topics

- Network Analysis techniques
  - Capturing network traffic
  - Using wireshark
- In-depth analysis techniques
  - Carving techniques/algorithms
- Custom binary parsing
  - Common binary layouts
  - Aside: Secure parsing techniques

data recovery: “live” sources

- Wiretaps, data hoses, live RAM on a seized PC
    - ...or just a file containing one of these
- Various tools/interfaces for generating these:
    - Net: Wireshark, `tcpdump`, a physical tap
        - Often saved in pcap format.
    - RAM: `/dev/{k,}mem`, `/proc/$pid/maps/`
        - `root` can read the whole address space!

data recovery: “live” sources

- We can collect all traffic on an interface with a single `tcpdump` command:

```
$ sudo tcpdump -w mydump.pcap -i eth0
```

Legend:

- `-w <file>` : the file to write the pcap to
- `-i <ifce>` : the interface to listen on
    - Check `ifconfig` to see your interfaces!

content-aware analysis: packet captures

- Packet captures often contain *all* traffic on a given interface over a period of time
  - A *lot* of data!
- To make sense of captures, we can use the filtering features of `wireshark` or `tcpdump`.
- Things we might want to filter for:
  - `http / https` - HTTP(S) connections
  - `ip.src == 1337 / ip.dst == 1337` - port traffic

packet capture analysis - live demo

Let’s tear a packet capture apart and see what kind of data we can find

*HW hint: Follow → TCP Stream

analysis: physical media

- The goal: extract as many files as possible from the image for later analysis
- The problem: lots of data, limited time and RAM
- The solution: carve the disk image
- Carving?
    - Extracting interesting signatures! (recall forensics 1)
    - Involves file format parsing

file carving algorithms

- (Naive) serial algorithm:
  - Inputs: *K* GBs of input data, *1* worker
  - Give the worker all *K* GBs, either as a big array of bytes or as a (seekable) input stream
  - Worker scans the entire input for signatures
  - Worker attempts to extract (or at least report) the correctly sized chunk of data for each signature
  - Pros: Simple to implement, fast for small *K*
  - Cons: CPU bound, slow for large *K*

file carving algorithms

- (Naive) parallel algorithm:
  - Inputs: *K* GBs of input data, *N* workers
  - Give each worker *K/N* GBs of the input
  - Each worker scans their section of the input for signatures
  - Each worker attempts to extract (or at least report) the correctly sized chunk of data for each signature
  - Not I/O bound, so we can use threads!
  - **There's a major problem with this algorithm!**
    - What happens when a file of interest gets split between two workers?

file carving algorithms

- (Better) parallel algorithm:
  - Inputs: *K* GBs of input data, *N* workers
  - Give each worker *K/N* GBs of the input
  - Assign each worker a specific magic byte to look for
  - Each worker scans their section of the input for *their* signature
  - Each worker attempts to extract (or at least report) the correctly sized chunk of data for each signature
  - Possibly use GPUs to speed up
  - **There are drawbacks to this approach!**
    - https://www.dfrws.org/sites/default/files/session-files/pres-massive_threading_-_using_gpus_to_increase_the_performance_of_digital_forensics_tools.pdf

interesting signatures

- Signatures come in all forms, but we’re specifically interested in file signatures
  - But remember the others: emails, domains, &c
- Magic bytes are the most common file signature
  - A (mostly) fixed sequence of bytes that identify a particular type of file (PDF, JPEG, EXE)

magic bytes

- Magic bytes are at the beginning of a file*
- Examples of file magics:
  - ZIP: \x50\x4B\x03\x04 (PK..)
  - ELF (Linux exec): \x7F\x45\x4C\x46 (.ELF)
  - MZ (Windows exec): \x4D\x5A (MZ)
  - Mach-O (macOS exec): \xCA\xFE\xBA\xBE
  - PDF: \x25\x50\x44\x46 (%PDF)

* usually (some formats use footers)

magic bytes


%PDF-1.4.%.... 5 0 obj.<</Length 6 0 R>>.stre
am.0.09375 w.0. 1 0.216 0.541 RG.0.11 0.216 0
.541 rg.0.00 30 .00 m.228.00 305.00 l.228.00
0.00 l.0.00 0 0 l.0.00 305.00 l.f.0.875 0.74
9 0.741 RG .875 0.749 0.741 rg.70.53 260.03
266.70 71.59 274.74 76.79 280.16 c.82
.07 275.92 85.56 269.74 91.65 266.48 c.95.82
267.69 100.17 267.18 104.33 266.27 c.104.60 2
64.68 104.75 263.06 104.63 261.45 c.103.27 26
0.08 101.92 258.65 100.11 257.89 c.101.64 259
.75 103.50 261.72 102.97 264.35 c.98.70 264.6
3 94.42 264.35 90.16 264.07 c.88.95 263.89 87
.65 262.95 86.45 263.49 c.82.97 267.98 81.60
274.02 76.87 277.54 c.75.53 271.35 73.02 265.
41 72.64 259.05 c.70.21 254.27 65.14 250.89 6
4.94 245.13 c.61.66 242.22 60.09 238.07 58.16
234.25 c.61.95 232.59 66.07 233.45 70.05 233
.25 c.72.49 233.28 73.83 230.94 75.46 229.51
c.77.91 226.92 81.79 226.01 83.68 222.84 c.86
.69 218.12 90.82 212.71 88.59 206.79 c.88.01
202.21 82.06 200.47 78.39 202.46 c.75.90 203.
55 73.89 206.39 70.89 205.62 c.67.86 205.21 6
4.77 205.06 61.79 205.90 c.58.53 206.96 55.30
205.34 52.03 205.24 c.48.69 205.67 45.47 207
.29 42.04 206.83 c.39.35 206.31 36.86 205.09
34.30 204.15 c.35.49 198.49 38.62 193.49 39.7
0 187.79 c.40.08 185.09 43.15 184.13 44.19 18
1.85 c.44.91 178.43 44.82 174.76 46.79 171.71
c.48.70 168.59 48.56 164.83 49.66 161.43 c.5
1.35 156.07 50.52 150.35 51.54 144.87 c.52.74
144.65 53.95 144.43 55.15 144.20 c.55.56 141
.24 56.59 138.34 59.12 136.54 c.59.62 134.93
60.11 133.31 60.59 131.69 c.63.91 135.10 67.4
3 138.32 71.37 141.01 c.71.86 141.09 1.73.08
140.22 1.69.17 138.13 64.99 135.55 63.20 131.
30 c.62.60 130.12 61.01 129.98 59.99 129.29 c

magic bytes

- Why are file magics used?
  - File extensions are easy for users to modify, so trusting them can result in attempting to parse the wrong format
  - Provides us a way to version file formats: users don’t care about GIF87a vs. GIF89a, they just want their `.gifs` to open
  - **Not** a panacea: formats can have conflicting magics (see: Java classfiles and Mach-O)
    - Heuristics required!

file extraction

- We need to know how large a file is/where it ends in order to extract it from the image!
- Techniques vary by file format:
  - Look for “footer” bytes (\xFF\xD9 for JPEG)
  - Attempt to calculate the file’s size
    - Involves parsing a subset of the file
  - Heuristics:
    - Trim files after *K* MBs, note offset for later retrieval if the file isn’t correct

binary parsing

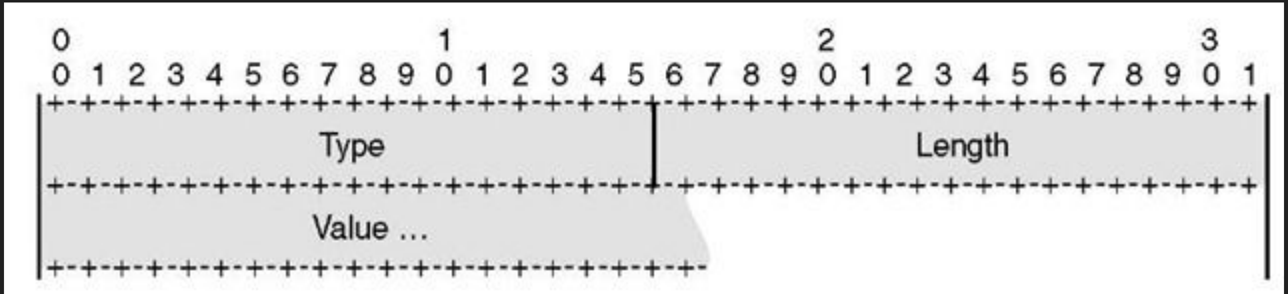
- Two views: carving view and “correctness” view
  - Carving: Just enough to extract correctly
  - Correctness: Understand the whole file
  - We usually end up with a mix of the two
- Unusual formats commonly pop up during forensics:
  - Camera-specific “RAW” photos (often TIFF-based)
  - Custom application formats (intermediate files)
  - Ancient/insane formats like pre-03 `.docs`
    - OLE Compound Files

binary parsing: techniques

- Most binary files are divided into (at least) two components: a header and a body
- Header contains metadata (file magic, version, ...)
  - Usually fixed-length, e.g.: 4 bytes for magic, 4 for version, 8 for UUID, etc.
  - Can contain parsing directives (endianness, how much data to expect)
- Body contains file content (photo data, ...)
  - Often variable-length, with discrete sections tightly packed together

binary parsing: techniques

- Many formats use a type-length-value (TLV) layout
- Type and length fixed at *K* bytes, value variable
  - Type indicated by a number, which gets looked up in a table (1 for text, 2 for array, etc…)
- Does the length field give you the length of value, or value + 2*K*?
  - Depends on the format!



binary parsing: techniques

- How do we actually write a binary parser?
- Many scripting languages provide `pack/unpack`:
  - `array.pack(format)` -> binary string
  - `binstr.unpack(format)` -> array
  - Options for endianness, padding, …
- C makes structure packing/unpacking “easy”:
  - Packing: write `sizeof(type)` bytes to the stream
  - Unpacking: cast the incoming bytes + offset to your type
  - Don’t use C for this week’s assignment!

aside: secure binary parsing

- Parsers need to be *resilient*:
  - Shouldn’t crash on weird inputs (exit gracefully)
  - Shouldn’t hang (detect and avoid cycles)
    - ZIP bomb*
  - Shouldn’t leak memory (buffer overruns, overflows)
- Many security problems stem from *trusting input*:
  - Trusting specified length instead of the spec
    - Specs often limit values to a maximum length!
  - Not checking whether a specified length is beyond the input buffer

“... 1.3GB file full of zeroes, compress that into a ZIP file, make 10 copies, pack those into a ZIP file, and repeat this process 9 times … when uncompressed completely, produces an absurd amount of data without requiring you to start out with that amount.”

homework #9

Will be posted.

Let us know if you have any questions!

This assignment has 2 parts.

It is due by 11/11 at 11:59PM.