Introduction to InfoSec – Recitation 6

Nir Krakowski (nirkrako at post.tau.ac.il)
Itamar Gilad (itamargi at post.tau.ac.il)

Today

- Introduction to fuzzing
- Fuzzing principles
- Useful tools and leads

What is fuzzing?

- Fuzzing == Looking for vulnerabilities not by reverse engineering the binary / source code, but rather – by playing with the data
- ובעברית == התססה

- The basic idea is to throw lots and lots of 'weird' input at the target, and see if things break
- Assuming the target is poorly written, bad data will crash it
  (Even if it won't crash, we can usually find a way to know if something bad happened)

Fuzzing principles

- Fuzzing takes time
- We want to automate everything
- Basic elements in our 'endless' loop –
  - Target startup (process / network service / VM)
  - Feeding the target our test vector
  - Monitoring the target's health (black box / white box)
  - Logging successful iterations – 'bad' test vector (and the way it was generated if relevant)

Generating test vectors

- Naïve solution: iterate over the entire input range
- But that is impractical for almost all scenarios
- Our objective is to find a way to cover the most lines of code in the least amount of CPU time

- Two main approaches –
  - Generation – generate input from scratch, following the most minimal set of rules required
  - Mutation – Using a valid starting point (sample), start flipping / moving / inserting bits

Preparing to fuzz a format

- Read the protocol / file format documentation
- Observe several test-cases with ghex / wireshark
- Find important fields –
  - Length fields
  - Checksums
  - Flags
  - Tree-like structures
  - Etc.
- We may need to make sure some or all of these are 'correct'
- **Or, we may want them to be incorrect!** ☺
-

Basic example - Browser

- We'll try to fuzz MS Internet Explorer 6.0
- Elements –
  - Target startup + Feeding the target our test vector by running "iexplore %s" %target_file
  - Monitoring the target's health – by testing if the process is still alive after 'x' seconds
  - Any sample that crashed IE will be saved
  - Before moving to the next iteration – kill left-over IE to always start with similar conditions

Another example – Network Daemon

- Almost any network server will do
- Elements –
  - Feeding the target our test vector via a TCP connection
  - Monitoring the target's health – by issuing benign requests constantly
  - If the server has died – log Everything
  - Collect core dump via ssh / remote agent
  - Restart via ssh / remote agent

Faster? Stronger?

- Until now, we stuck with the most external level of control
- We want to have more information about the crash
- And we want to be able to try more test vectors
- But we don't have more CPU time…

Faster! Stronger!

- Solution: restart from the middle –
  - Install a breakpoint
  - ‘Freeze’ the memory and state when the breakpoint is triggered
  - Apply test vector
  - Run until function / module / process exit / crash
  - Restart
- Can be done
  - With a ‘smart’ debugger
  - With an entire VM (Leveraging snapshots)
  - With better instrumentation (white-box testing)

The sad truth

- The truth is that finding crashes is not that hard
- Finding **exploitable** vulnerabilities is **a lot** harder
- Good fuzzing usually requires a lot of tuning and after-processing to filter out the noise
- After you get a crash, there are some Triage steps that can help you figure out "how exploitable" the bug is –
  - Form your data with simple patterns
  - Inspect registers, stack & heap after crash
  - See if any of your patterns remain
  - (slightly) modify the patterns and repeat

Further reading & Tools

- Google ‘fuzzing’ / ‘fuzzer’ / ‘python fuzzer’ / ‘protocol fuzzer’
- Fuzzing.org is a great resource
- As are many good papers / slideshows you’ll find via google(especially lectures from BlackHat)

This week's exercise

- Fuzzing the ImageMagick image-processing suite
- Fuzzing Image file formats

- Mutation based fuzzing ("bit flipping")
- Bonus – Smarter fuzzing ☺

Questions?