8/31 - CS261 Lecture 3 - Fuzzing and Symbolic Execution

by: Casey Lawler

More Attacks (Wrapping up last meeting's discussion on Buffer Overflows)

JIT Spraying:

- JIT spraying is an attempt to circumvent the Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) defenses.
- In a JIT spraying attack, the attacker "sprays" memory with many instances of shellcode in memory regions marked as executable for the output of Just in Time (JIT) compilers.
- Attackers will often employ NO-OP "landing pads" or "slides" to increase their likelihood of success.
- Browsers are high-profile programs that tend to have great potential for buffer overrun vulnerabilities (more use and SLOC → great targets) -- these were often targeted with JIT Spraying attacks along with PDF viewers and other programs.

Return-Oriented Computing: "A Beefed Up Version of Return-to-libc"

- There must exist a buffer overrun on the stack for this attack to occur.
- In return-oriented computing, attackers set up fake stack frames and jump into existing code sections by overwriting return addresses.
- The main insight: don't inject full functions to execute -- jump to *any* part of an executable code section with custom arguments and use these code segments to execute arbitrary processes.
- With high probability, the set of instructions an attacker might use in the executable code segments is Turing complete.
- This attack is *very* methodical and must be carefully-crafted and can be defeated with ASLR.

Main Takeaways Regarding Buffer Overruns:

- Constructing and efficiently traversing shadow data structures (as in Baggy Bounds checking) with bounds information is an effective measure to prevent buffer overruns.
- When analyzing different execution paths (that diverge at branch instructions), it is very common for there to be "fast paths" taken with high probability and "slow paths" taken with low probability.
- Buffer overrun protections seem to focus on instrumenting source code. Analyzing binaries tends to be more difficult due to compiler optimizations making use of all available registers (bounds checking tends to use at least one other register).
- DEP, ASLR, and StackGuard (w/ canaries) are widely deployed as buffer overrun protections today. Exactly which defenses are deployed is always a debate between performance and security.

Whereas the preceding points regard defenses that guard at run-time, the following tools are "proactive" bug finders.

Fuzz-Testing:

- Fuzz-testing is the practice of generating inputs (generally within some constraints) and running the program under examination with these inputs to observe its behavior.
- Inputs can be generated as follows:

    - *Randomly* -- bad because generally these inputs don't pass the program's first validation tests (like checksums)
    - *By Mutation of Valid Inputs* -- many generated inputs are well-formed and pass the program's first validation tests
    - *Generationally* -- randomization with knowledge of the file format
- Fuzz-testing is simple to perform, computationally cheap to accomplish, and is surprisingly effective.
- A common coverage metric for fuzz-testing is how far a particular case gets through the program to be tested. By generating and preferentially mutating inputs with high code coverage, the hope is that more of the program under examination can be tested.

EXE -- Symbolic Execution:

- EXE utilizes built-in bounds checking with shadow data structures like baggy bounds checking.
- Symbolic execution tends to be much more computationally expensive compared to fuzz-testing; as a result, code tends to be fuzz-tested first and analyzed via symbolic execution afterwards.
- Symbolic Execution programs exist that work with binaries as well as with source code -- one such program called Sage was developed by Microsoft.
- Symbolic execution programs like EXE can be used to find inputs that crash one program and not another (like Apache vs Nginx). This is called *program fingerprinting*. These inputs are likely to be bugs. EXE could find them by running a program as follows:

```
make_symbolic( &x );
y = apache( x );
z = lighthttpd( x );
assert( y == z );
```

- Given a patched version of a program and an unpatched version of the same program, EXE can be used to *derive exploits* by running similar code to that in the block above (just replace apache(x) with new_version(x) and lighthttpd(x) with old_version(x) to see this).