Secure Software Design

Andey Robins

Spring 23 - Week 13

MAC Assignment Retrospective

Common Problems

1. Writeable database fields
2. Hashes instead of HMACs
3. Crashing out on bad MAC

Writeable DB Fields

Almost always possible:

```
UPDATE Messages SET data="hax" WHERE id=1;
```

Often impossible:

```
UPDATE Messages SET hmac="hax" WHERE id=1;
```

How To Prevent Updates

```
CREATE TRIGGER mac_update
BEFORE UPDATE OF hmac ON Messages
BEGIN
    SELECT raise(abort,
        "Attempted to alter the hash of a message."
    );
END
```

Hash vs HMAC

- Hash is just a digest associated with an input value. Generated with a one-way-function.
- HMAC incorporates a private key, ensure the authenticity of the creator of the HMAC.

Crashes

Crashes are bad, yet very understandable. It's important to test your code not just in the ways it is supposed to work, but also in ways with a bad actor present.

Security Testing

Outline

- What is security testing
- Types of security testing
- Limitations
- Regression and availability tests
- Best practices

Security Testing

Most testing consists of exercising code to check that functionality works as intended. Security testing flips this around, **ensuring that operations that should not be allowed aren't.**

Testing Targets

- Integer overflow/underflow
- Memory management problems
- Untrusted inputs
- Web security
- Exception handling flaws

Testing Targets

- Integer overflow/underflow (Wk 10)
- Memory management problems (Wk 10)
- Untrusted inputs (Wk 11)
- Web security (Wk 12)
- Exception handling flaws (Wk 8)

Normal programming is all about getting things to work as intended. Security testing just validates that is the only thing possible with software. Another way to look at it would be checking a boat for leaks before putting it into the sea.

GoTo Fail

We've seen this previously two other times.

```
if ((err = SSLHashSha1.update(&hashCtx,
        &serverRandom)) != 0)
    goto fail;
    goto fail;
```

instead of

```
if ((err = SSLHashSha1.update(&hashCtx,
        &serverRandom)) != 0)
    goto fail;
```

Testing GotoFail

Functional Testing: We ensure everything actually works as expected. (assume that the Goto fail vulnerability isn't there)

```
mu_assert(0 == VerifyServerKeyExchange(test0,
    expected_hash,
    SIG_LEN),
    "Expected correct hash check to succeed.");
```

Functional Testing with GotoFail Present

Due to the structure of the GotoFail vulnerability, this valid test will still pass. However, it will now also verify inputs where the third argument is bad.

Functional testing often includes all of the positive cases, but often does *not* include all of the non-functional branches.

Security Testing

The values of `test1`, `test2`, and `test3` are the same as in `test0` except for one of the fields has been corrupted.

```
mu_assert(-100 == VSKF(test1, ...),
    "Expected to fail: wrong client random.")
mu_assert(-100 == VSKF(test2, ...),
    "Expected to fail: wrong server random.")
mu_assert(-100 == VSKF(test3, ...),
    "Expected to fail: wrong signed param.")
```

Results with Vulnerability Present

All three of these tests happen to find the GotoFail vulnerability. In many cases however, only a single security test would fail to indicate some sort of security concern.

Testing Input Validation

It is impossible to test every single potential valid input.

Assume we want to test code which requires input be: - alphanumeric (ascii) - at least 10 characters - less than 20 characters

Testing Input Validation

Instead of exhaustive testing:

- Test failure of 9 or less and acceptance of 10
- Test failure of 21 or more and acceptance of 20
- Test inputs with an invalid character anywhere fail
- Test a valid input

Testing for XSS

1. Write code which attempts to properly escape all potential special html characters
2. Write simple unit tests which verify the code
3. Write extended security tests which leverage a library to verify the code
4. (optional) Fuzz the test with a known XSS corpus

Fuzz Testing

Fuzz Testing is a technique which automatically generates test cases in an effort to find breaking cases. Breaking is defined in terms of some programmer supplied invariants in the test.

```go
// setup
func FuzzReverse(f *testing.F) {
    testCases := []string{
        "hello world",
        "  ",
        "123567!@#",
        "this is a sentence",
    }

    for _, tc := range testCases {
        f.Add(tc)
    }

    /// --- SNIP --- ///
}
```

```
/// --- SNIP --- ///
// continued, error checking omitted
f.Fuzz(func(t *testing.T, in string) {
    got, err := Reverse(in)
    doubleGot, err := Reverse(got)

    if doubleGot != in {
        t.Errorf("got %q, want %q", doubleGot, in)
    }
    if !utf8.ValidString(got) &&
        !utf8.ValidString(doubleGot) {
        t.Errorf("got invalid utf8 string: %q", got)
    }
})
```

Fuzzing Guides

- JS/TS - Jest
- JS/TS - Mocha/Chai
- Go
- Rust

Limitations

- Security testing will never cover all of the possible ways code can go wrong
- Vulnerabilities might be introduced which existing tests don't cover

Rules of Thumb

- Security critical code is the most important to test
- Most important to check are places where you deny access, reject input, or otherwise fail
- Should also verify each key action succeeds when appropriate too

Security Regression Tests

Availability Testing

Resource Testing

Threshold Testing

Distributed Testing

Best Practices

Exception Testing

Assorted Best Practices

Documenting Security

Dependency Management

Legacy Security

Vulnerability Triage

Crafting Exploits

Secure Your Tools

Questions?

Next Time

- 2 weeks of no class
  - Project work time
- Group presentations
  - Last week of class