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 --- ///
}
```

```go
/// --- 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

First, what is regression testing?

Regression testing is testing which validates old bugs are no longer present in the latest version of the software.

Security regression testing is the same principle applied to security vulnerabilities

Security Regression Testing

1. Write a secure regression test which demonstrates the proof of concept of the attack
2. Write a fix to mitigate the security vulnerability
3. Continue to check that test as you perform updates in the future

iOS 12.4

In iOS 12.3, there was a simple vulnerability which allowed for easily jailbreaking the OS. It was patched, but when 12.4 was released, the vulnerability re-emerged.

Why is this worse than a new vulnerability?

Old vs New Vulnerabilities

New vulnerabilities are relatively unknown. They may not have a critical mass of knowledge and are unlikely to have complex, highly-developed toolchains.

Old vulnerabilities likely do have well known toolchains, the attack vector is better understood, and people are aware of how to craft actionable exploits for the vulnerability.

Even More Selling Points

- You understand the scope of an exploit better if you must test it
- You can apply other testing strategies to find similar vulnerabilities
- Horizontal pivoting can also be testing across related components very easily

Security Testing for Heartbleed

Heartbleed is the one where you can get arbitrary amounts of data back from a "heartbeat" webserver endpoint. The correct behavior as defined by the RFC is to ignore such malicious requests.

1. Test a known exploit is no longer answered
2. Test with request byte count larger than the maximum
3. Test with payloads of size 0 and max size
4. Check other packet types in the TLS spec for similar buffer overflow

Availability Testing

DoS attacks are all about overwhelming the “load” of the server where load refers to any of the following:

- processing power
- memory consumption
- OS resources
- network bandwidth
- disk space
- entropy pools
- etc.

Availability Testing

Availability Testing is all about testing code in these extreme situations to make sure it performs gracefully.

Beyond the approaches we'll discuss, some special purpose tools exist for the purpose as well:

- Hping3
- Goldeneye
- Hulk
- Slowloris

Resource Consumption Estimates

Placing your server under 100% load every time you want to do testing is probably not feasible. Instead, we can perform some estimates to determine resource consumption.

1. Run tests for input/data of size (N) where n is whatever measure of resource is worth considering.
2. Run the same tests for size (N + 1)
3. Extrapolate the maximum allowable input and determine if it fits within necessary thresholds

Backtracking Regex Test

```
def backtrack_match(s):
    re.match(r'(D+)+$', s)


def time_backtrack(n=1):
    start = time()
    backtrack_match('D'*n + '!')
    return time() - start
```

Heuristically Estimate Upper Bound

```python
def unit_test():
  MIN_SUPPORTED_SIZE = 50
  MAX_SUPPORTED_TIME = 600
  X, y = [], []
  for i in tqdm(range(22, 27)):
    X.append(time_backtrack(i))
    y.append(i)

  model = np.polyfit(X, np.log(y), 1)
  prediction = np.exp(model[1]) * \
    np.exp(model[0] * MIN_SUPPORTED_SIZE)

  if prediction > MAX_SUPPORTED_TIME:
    print(
      f'Estimated time of {prediction} was greater \
        than limit {MAX_SUPPORTED_TIME}')
```

Fixes

We identify the backtracking to be a problem, so refactor to:

```
def backtrack_match(s):
  re.match(r'(D+)$', s)
```

We set the constants appropriately, and now our test will fail if we somehow re-introduce backtracking to the regex!

Threshold Testing

Threshold Testing is a different type of “test” in that it will test the system against some resource limit and throw up a warning when things get to a pre-defined point.

For example, having a warning set up for when your disk reaches 80% usage.

Trick: Ballast

One simple little trick to allow you to quickly be able to respond to a server running out of storage is to keep a “ballast” file, which is a large file full of nothing which can be deleted in case of an emergency to eek out some extra time.

This is not recommended by many, but it’s a nifty trick to be aware of for dire circumstances.

Distributed Testing

How do you, as a single developer, test against a DDos attack?

Distributed Testing

In some situations, where many resources are available, you may be able to build a “mock network” where your system runs and you can do your testing on this simulacrum of the distributed network.

Example: Ganache and testing for ethereum.

Penetration Testing

Moving through types of availability testing, we arive at perhaps one of the most obvious: **penetration testing**.

Not many locations have the resources to perform these tests, but hiring outside hackers to try to break into your system is one of the easist ways to find out if there are any vulnerabilities.

Exception Testing

Exception Testing could be considered as a specific type of integration testing. When exceptions are thrown, where are they handled? Is all of the code in between the thrower and catcher capable of handling any of the possible exception values they are working with?

Best Practices

- TDD
- High test coverage
- Integration testing
- Focus on high security locations and work down

Assorted Best Practices

Documenting Security

When code has security implications, they are likely documented in a few places (i.e. design doc, project issues, internal messages, etc.). Make sure at least one of those places is the code itself.

```
// beware: security implication in this function
```

instead do

```
// authentication code: refer to issue #177 for discussion
// trust boundaries. code must only be invoked by authentic
// and authorized users. see auth middleware in //middlewar
```

Dependency Management

As we've said repeatedly, make use of the libraries instead of writing code yourself whenever possible. This adds the additional concern of then vetting external code before adding it to your project. This problem is known as dependency management.

Choosing Secure Components

This returns to the question of Trust:

- What is the security track record of this component/library/framework?
- What is the track record of the developers working on it?
- What alternatives are there?
- When vulnerabilities are found, are you confident in developers quickly fixing it well?
- What are the operational costs of maintaining the use of this component?

Legacy Security

- Security policies must be regularly updated to maintain relevance
- Older components should be regularly evaluated and their replacement/update considered
- Inter-operability problems may prevent the “best” option
- Once some security tool is set up, it’s difficult to re-consider it as being a potential attack later

Vulnerability Triage

General priority list:

- Bugs in privileged code or code that accesses valuable secrets
- Bugs in known vulnerability chains
- Bugs which cannot be easily assessed for security implications
- Bugs which appear to have no security problems

Crafting Exploits

1. Start with well known attack categories and attempt to apply them to the code in question
2. Make rigorous use of all the penetration methods you know (e.g. fuzzing, code analysis, etc.)
3. Build on known bugs to look for vulnerability chains
4. Practice finding exploits in known vulnerable libraries

Secure Your Tools

Don't forget that your tools are a key part of the security of your work. Would you trust a surgeon with rusty tools?

- Securely configure your ecosystem so that breaking trust cannot be done accidentally
- Regularly update all toolchain components (preferably automatically)
- Restrict personal computer usage where possible
- Review all new components before adoption
- Lock-down computers in the build/release pipeline
- Maintain secure credentials
- Regularly audit computer logs, commits, and systems
- Maintain secure backups

Questions?

Next Time

- 2 weeks of no class
    - Project work time
- Group presentations
    - Last week of class
    - Start May 1st