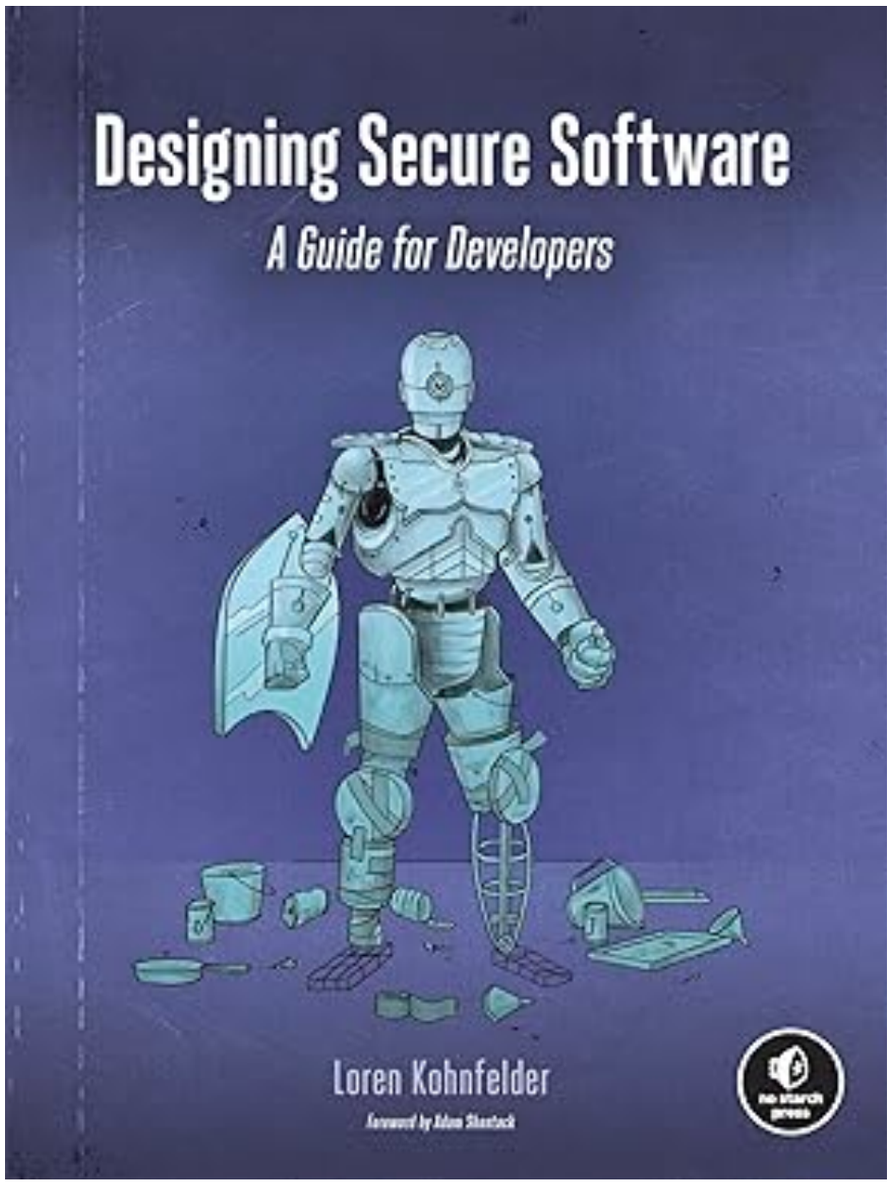


12 Security Testing

Topics

- What is Security Testing?
- Security Testing the GotoFail Vulnerability
  - Functional Testing
  - Functional Testing with the Vulnerability
  - Security Test Cases
  - The Limits of Security Tests

Topics

- Writing Security Test Cases
- Fuzz Testing
- Security Regression Tests
- Availability Testing
- Best Practices for Security Testing

What is Security Testing?

Security Testing

- Most testing checks that functionality works as intended
- Security testing ensures that operations that should **not** be allowed aren't
- Types of security test cases
  - Integer overflows
  - Memory management problems
  - Untrusted inputs
  - Web security
    - HTTP downgrade, CSRF, XSS
  - Exception handling flaws

Security Testing the GotoFail Vulnerability

Complete Function

```
int VerifyServerKeyExchange(ExchangeParams params,
   uint8_t *expected_hash, size_t expected_hash_len) {
   int err; HashCtx ctx = 0;
   uint8_t *hash = 0;
   size_t hash_len;
   if ((err = ReadyHash(&ctx)) != 0) goto fail;
1  if ((err = SSLHashSHA1.update(ctx,
      params.clientRandom, PARAM_LEN)) != 0) goto fail;
2  if ((err = SSLHashSHA1.update(ctx, params.serverRandom,
      PARAM_LEN)) != 0) goto fail;
   goto fail;
3  if ((err = SSLHashSHA1.update(ctx, params.signedParams,
      PARAM_LEN)) != 0)  goto fail;
   if ((err = SSLHashSHA1.final(ctx, &hash, &hash_len)) != 0) goto fail;
   if (hash_len != expected_hash_len) {
      err = -106;
      goto fail; }
4  if ((err = memcmp(hash, expected_hash, hash_len)) != 0) {
   err = -100;  // Error code for mismatch }
   SSLFreeBuffer(hash);
fail: if (ctx) SSLFreeBuffer(ctx); } return err;
```

Simplified Function

- Extra **goto fail** line makes it skip the SSL certificate verification test
- Functional testing will pass--it works for valid SSL certificates

```
1 if ((err = SSLHashSHA1.update(ctx,
     params.clientRandom, PARAM_LEN)) != 0) goto fail;
2 if ((err = SSLHashSHA1.update(ctx, params.serverRandom,
     PARAM_LEN)) != 0) goto fail;
   goto fail;
3 if ((err = SSLHashSHA1.update(ctx, params.signedParams,
     PARAM_LEN)) != 0)  goto fail;
4 if ((err = memcmp(hash, expected_hash, hash_len)) != 0) {
   err = -100;  // Error code for mismatch }

fail: return err;
```

Security Test Cases

- Try various types of invalid certificates and other invalid values
- Confirm that they are all rejected
- **Rules of thumb for security tests**
  - It's more important for security-crucial code
  - Most important: check for denying access, rejecting input, or otherwise failing
  - Ensure that each of the key steps work correctly

Writing Security Test Cases

Security Test Cases

- Create them along with other unit tests
  - Not only as a reaction to finding vulnerabilities
- Code must block improper actions, reject malicious inputs, deny access, etc., like:
  - Login fails with wrong password
  - Accesing kernel resources from user space fails
  - Invalid digital certificates are rejected

Testing Input Validation

- Input field should be alphanumeric, 10-20 characters
  - Check that a valid input of length 10 works, but an input of length 9 or less fails.
  - Check that a valid input of length 20 works, but an input of length 21 or more fails.
  - Check that inputs with one or more invalid characters always fail.

XSS Example

- User submits a color parameter

```
https://www.example.com/page?color=green
```

- Page applies that color to some text

```
<h1 style="color:green">This is colorful text.</h1>
```

vulnerable code

```
query_params = urllib.parse.parse_qs(self.parts.query)
color = query_params.get('color', ['black'])[0]
h = '<h1 style="color:%s">This is colorful text.</h1>' % color
```

XSS Example

- This is the attack string

```
https://www.example.com/page?color=orange"><SCRIPT>alert("Gotcha!")</SCRIPT><span
%20id="dummy
```

- Resulting HTML

```
<h1 style="color:orange">  <SCRIPT>alert("Gotcha!")</SCRIPT>  <span id="dummy">This
is colorful text.
</h1>
```

Testing for XSS Vulnerabilities

- If output contains user-controlled input values
  - Test for special characters like **<>"**
- The function below tests to make sure the parsed HTML structure is the same when a parameter contains a **"**
- Adds a <meta> tag to the end of the document
  - Verifies that it's found and unaltered

```
def test_parsed_html(self):
  for content in ['x', 'x"']:
    result = html_tag('meta', {'name': 'test', 'content': content})
    soup = BeautifulSoup(result, 'html.parser')
    node = soup.find('meta')
    self.assertEqual(node.get('name'), 'test')
    self.assertEqual(node.get('content'), content)
```

Fuzz Testing

- Sending many input values to see if they cause errors
- This function tries many punctuation marks to see if any of them alter the parsed structure of the HTML page

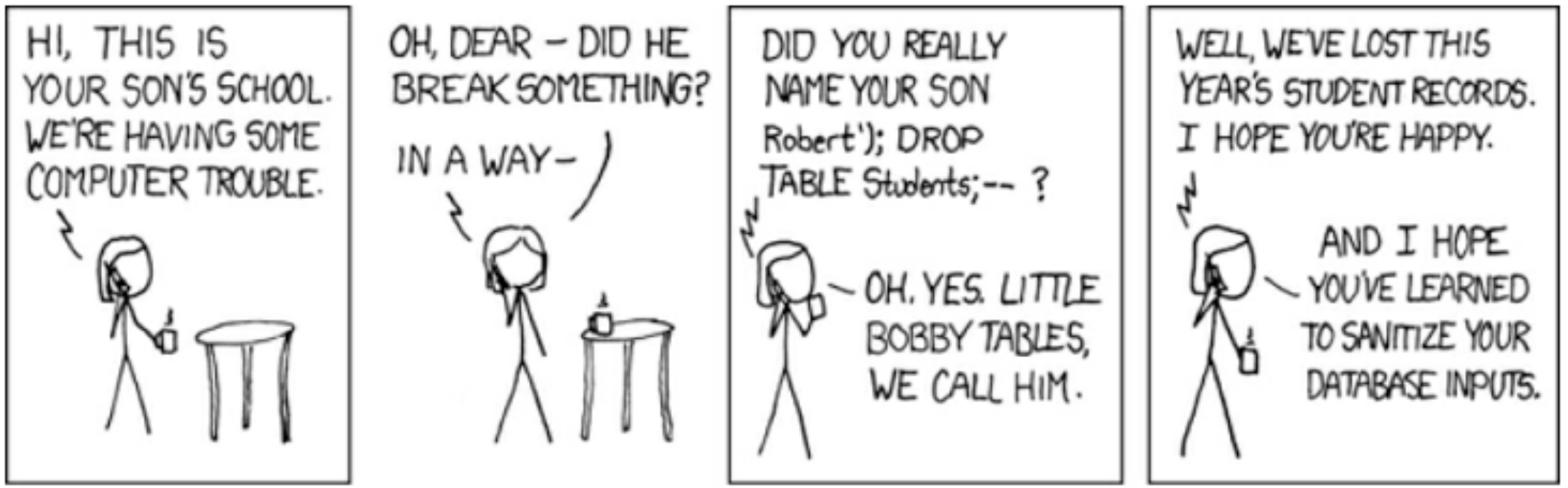
```
def test_fuzzy_html(self):
  for fuzz in string.punctuation:
    content = 'q' + fuzz
    result = html_tag('meta', {'name': 'test', 'content': content})
    soup = BeautifulSoup(result, 'html.parser')
    node = soup.find('meta')
    self.assertEqual(node.get('name'), 'test')
    self.assertEqual(node.get('content'), content).
```

Security Regression Tests

Security Regression Tests

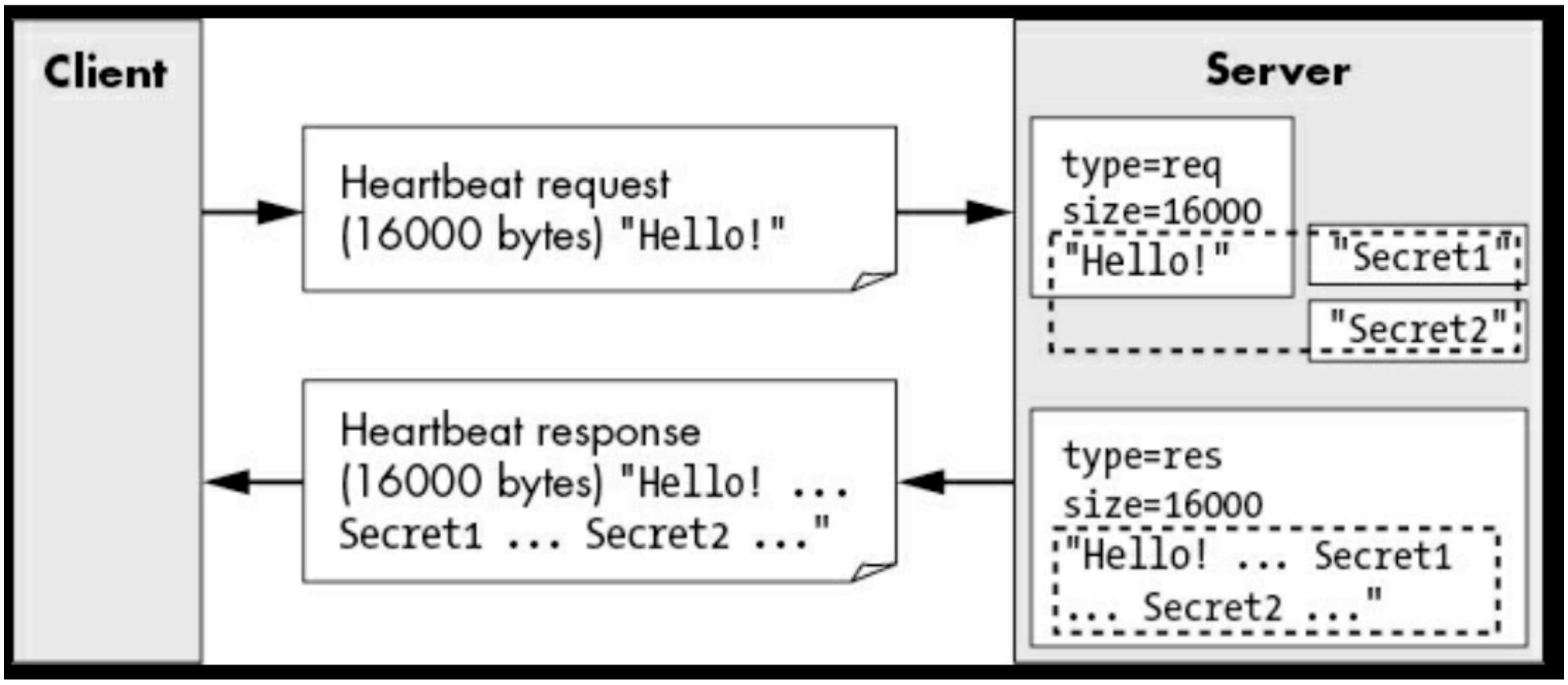
- **Regression testing** is performed after every update to ensure that the product is working correctly
- When a security bug is fixed, it often comes back in a later update
- Solution: create a **security regression test** that detects the bug
- Recommended: write the test case first, before writing the fix
- Try more than a single test case, be more general

SQL Injection Tests



- Don't just test **Robert';)**
- Try:
  - Long strings
  - **"** instead of **'**
  - **** at end of name, etc.

Heartbleed



- Test that known exploit requests no longer receive a response.
- Test with request byte counts greater than 16,384 (the maximum).
- Test requests with payloads of 0 bytes and the maximum byte size.
- Investigate whether other types of packets in the TLS protocol could have similar issues, and if so test those as well.

Availability Testing

Denial of Service (DoS)

- Security testing should include test cases to identify code
  - With nonlinear performance degradation
- DoS vulnerabilities

Resource Consumption

- Use security test cases to determine a sensible limit on the rate of input
    - Then perform input validation to prevent larger inputs from overloading the system
- An easy test is to use a system with artificially low memory available

Threshold Testing

- Establish warning signs before limits are reached
  - Integer overflow
  - Storage capacity full
  - Expiration of certificates
  - Y2K bug
- Y2k38 bug
  - On Jan 19, 2038 the Unix epoch will overflow
  - Dates will change to 1970

Distributed Denial-of-Service Attacks (DDoS)

- Many attacking devices working together to overwhelm the target
- Not easy to mitigate in server software
- A matter for ISPs, load balancers, firewalls, etc.

Best Practices for Security Testing

Best Practices for Security Testing

- **Test-Driven Development**
  - Write tests concurrently with new code
    - Best to make the tests first
- **Integration Testing**
  - Ensures that the complete system works properly, with all units working together
  - Ex: log in, make sure the password doesn't appear in logs or anywhere else exposed

Best Practices for Security Testing

- **Security Testing Catch-Up**
  - A codebase lacking security tests
  - Divide the job into pieces with incremental milestones
  - Target the protection mechanisms and functional areas in order of importance
  - Review existing test cases

Ch 12