Domo Arigato, Mr. Roboto

Security Robots a la Unit-Testing

Seth Law - March 30, 2017 - Black Hat Asia 2017 White Paper

Abstract

Security testing is difficult, no matter who is doing it or how it is performed. Both the security and development industries still struggle to find reliable solutions to identify vulnerabilities in custom code, but sometimes make things harder than they should be.

Over the past 20 years, the security industry has defined application security testing tools as separate from the traditional QA toolset, although both approaches are similar. Send test data (or payloads, exploits) to an application and inspect the response for appropriate or inappropriate behavior. The one-size-fits-all approach for security testing during the software development lifecycle (SDLC) does uncover some security flaws, not all, and leaves something to be desired, as it does not pinpoint the exact file/function where a vulnerability exists. Fuzzing application parameters is a great first step, but requires additional research and work to fix or exploit any identified flaws. Additionally, the traditional approach may not discover regressions in application code with the same speed and precision that unit-tests would.

On the other hand, the unit-testing frameworks provided by programming languages and application frameworks often lack functionality necessary to perform security testing. A lack of coverage, test data, or even functionality reduces the overall effectiveness of a security unit-test. In addition, identification of many security vulnerabilities, including cross-site scripting, requires fully functional application stacks with presentation layers. If the unit-testing framework is missing any of these pieces, it is impossible to create a full security test suite.

Due to both the aforementioned limitations of unit-tests as well as traditional security approaches to software security, custom and specific security testing is often overlooked and is not instituted within the typical software security testing tool suite. As developers and security professionals, we can do better. A hammer is not the only tool in our belt, and a scanner is not the only way to find a security vulnerability. Using DevOps practices such as Test Driven Development (TDD) and Continuous Integration (CI), it is possible to overcome both security and development weaknesses around unit-testing and implement a custom security unit-test suite for any application.

This paper will address the current limitations of security unit-testing applications with existing tools and various frameworks. Next it will introduce a generic framework for creating security unit-tests for any application. Then it will review common strategies for building application security-specific unit-tests, including function identification, testing

approaches, edge cases, regression testing, and payload generation. In addition, it will demonstrate these techniques in Java Spring and .Net MVC frameworks using intentionally-vulnerable applications. Finally, it will introduce SPUTR (https://github.com/sethlaw/sputr), an open-source repository of security unit-testing payloads that can be used as a starting point for creating custom security unit-tests.

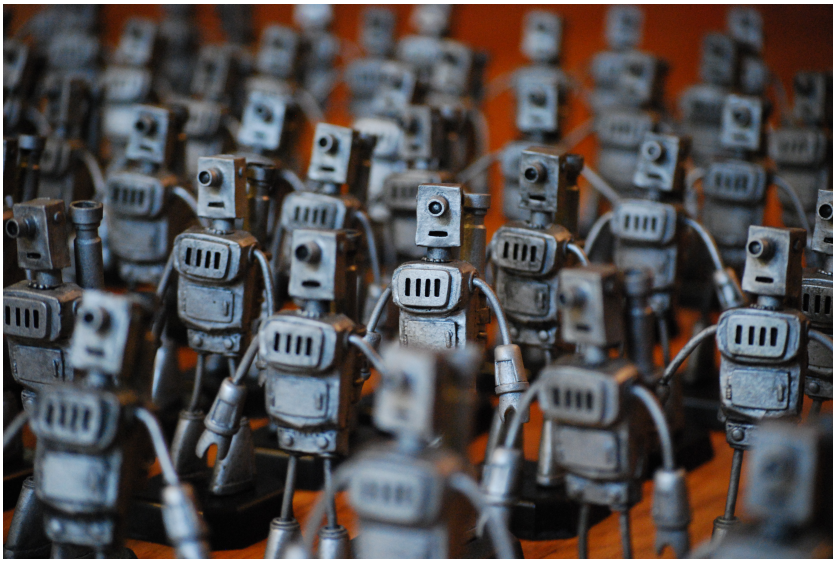
Introduction

Software developers have used various testing methodologies to ensure the quality of software products for years. A component of this process, unit and integration-testing concentrates on specific portions of the application and is used in different phases of the traditional Software Development Lifecycle (SDLC). Unit-testing typically occurs during the development phase, where developers validate that application functions respond with the appropriate output. On the other hand, integration-testing occurs during the testing phase and is used to validate that the different developed components interact appropriately.

Security has always been a development concern, but traditional security testing techniques have not always been taken into account during the aforementioned SDLC. This is in part due to lack of developer education, competing business objectives, and lack of proper testing tools. However, security vendors have attempted to bridge this gap by providing tools for both the development and testing phases of the SDLC, using cutting-edge security research to feed toolsets, payloads, and vulnerability identification.

A look at the daily security news comprised of leaks, hacks, and exploits shows that tools in this area are lacking. Security tools targeted at the development phase focus on insecure coding practices and patterns, while those targeting the testing phase concentrate on exploitation of vulnerabilities. These tools successfully identify security flaws, but do not go far enough to identify insecure practices or flawed functions. In addition, time and budget constraints restrict coverage and lead to flaws slipping through the cracks.

Combining these testing approaches can be used to further secure applications. By creating an army of testing bots with limited functionality, targeted payloads, and a deep understanding of application functionality, it is possible to uncover additional security flaws that the traditional tools may struggle with. Placing these bots in a continuous integration (CI) environment further increases the ability to test and identify these security flaws. This paper will demonstrate these techniques and introduce a framework for building these bots.



Current Security Testing Tools

As previously mentioned, the current suite of security testing tools target specific phases of the SDLC, specifically the development and testing phases of a typical waterfall methodology. These tools do uncover vulnerabilities, but have strengths and weaknesses as with any technology. Additionally, with the advent of continuous integration (CI) and test driven development (TDD) strategies, developers must understand how these security tools work and be able to implement them into the continuous deployment pipeline.

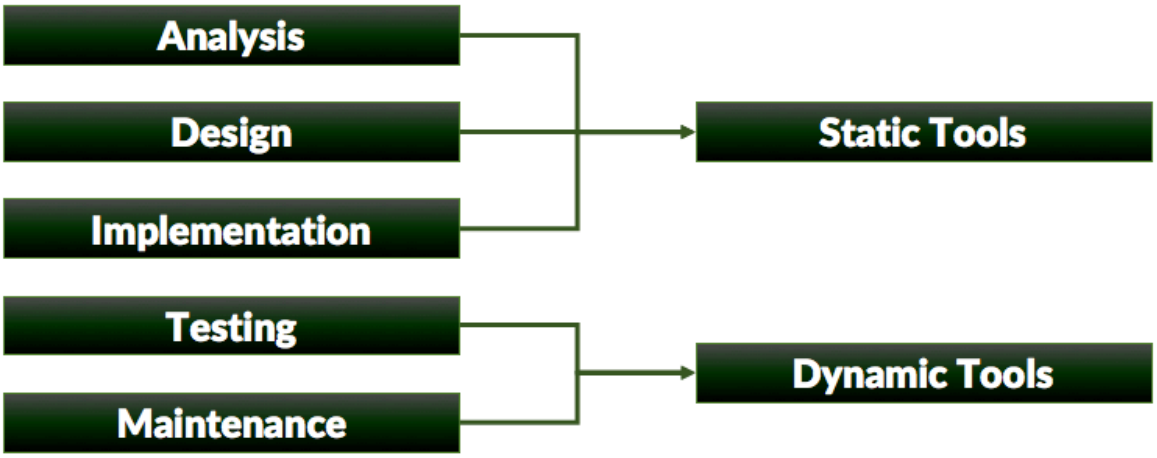
The name of the game with any security testing tool is vulnerability identification and false positive reduction. As anyone who has reviewed the results of these tools will tell you, the reports can be useful, but the amount of time needed to confirm that a vulnerability exists reduces the overall value of utilizing an automated tool. Furthermore, the cost of the tool increases beyond licensing fees, SDLC integration, and maintenance, as the expertise required to validate any findings usually exceeds entry level positions or those unfamiliar with the inner workings of the application.

Implementation of these tools can be technically easy, but as the author has seen, it is more difficult for a business to decide what should be done with the output. Questions around the results can gridlock a product and it's engineers. Each of the following questions must be answered, otherwise output will be ignored and resolution of any security issues succumbs to business pressures for delivering a functional, but possibly insecure, product.

- What is the process for remediating identified flaws?
- Who decides what is an acceptable risk?
- How far should we trust the criticality levels assigned by the tool?
- How well does the tool understand the technology it is analyzing?

While not a complete list, implementation of any third-party tool in the development process requires some business impact analysis. Sadly, most security tool implementation is driven by compliance and is never effectively integrated into a products development pipeline. Any identified security flaws must be analyzed and remediated after the product is already in production (or close to it) and causes unnecessary stress and financial investment to remediate.

To facilitate a conversation about the current landscape of security testing tools, we will discuss two different tool types, dynamic and static. Dynamic tools are defined as any security testing tool that interacts with a running application environment in order to identify any vulnerabilities. Alternatively, a static tool is one that inspects and instruments application source (or an equivalent thereof, such as byte code or compiled code with debug symbols) to uncover flaws. Some tool suites combine the approaches to gain further code coverage and false positive reduction, but flaws still occur even in these cases.



Dynamic Tools

The full range of dynamic application security testing tools ranges from open source (OWASP ZED Attack Proxy) to “freemium” (Burp Suite Professional) to fully paid (IBM AppScan Standard/HP WebInspect). Implementation of these tools into a CI or TDD pipeline is typically done by a build or developer operations (DevOps) engineer with input from the security team. The scans happen after the application is built successfully and determines whether known and discovered components contain security vulnerabilities.

In other words, dynamic security testing tools are implemented as a portion of a QA integration test, according to normal assurance testing methodologies. As such, they must be able to provide full coverage of all application components using some sort of a discovery or scanning process. This process is usually automated in some fashion, using spidering techniques or by training the tool in interacting with the application. Depending on the application, this process can be problematic due to authentication routines, authorization issues, and hidden application functionality. Without proper tuning and checks, it is possible that the tool is not exercising full functionality.

Static Tools

Static tools have become more popular in the last decade and also comprise of free and paid versions. The one difference is that most free options (Brakeman, Findbugs, etc) target a specific language or framework, whereas paid options (HP Fortify, IBM AppScan Source, Veracode) cover multiple languages. Implementation of these tools occurs in both the development and testing phase of a traditional SDLC by developers or build engineers. They can be used to validate secure functionality as soon as code is written and verifiable and most versions include IDE plugins that are used by developers to scan code for flaws during active development.

In terms of assurance testing, static tools are used for both functional and integration tests, where a single developer will run the tool against their individual components and the build process will contain a fully integrated application. This ability to run at a lower level gives static tools advantages over dynamic tools in terms of vulnerability identification and remediation advice, but also requires more effort by the tool vendor, which equates to a higher initial cost.

Tool Strengths

Dynamic and static testing tools provide a number of strengths, including speed of setup, cost, and ability to meet compliance needs. They are especially proficient at identifying generic vulnerabilities using known payloads. The first iteration of most security testing tools could be considered regular-expression engines, where the tool watches application output for known vulnerable output. Essentially just the same basic virus scanner or firewall with different input sources.

For example, Cross-Site Scripting (XSS) vulnerabilities are the result of flaws in user input validation and output encoding. They are well understood by the security community and payload lists for the vulnerability exist. A tool can spider a web application, identify any form fields and parameters, insert XSS payloads, and parse the output looking for the response.

Various dynamic tools have different methods for parameter detection and payload creation that help eliminate false positives, but the above strategy has been proven effective for positively identifying whole classes of vulnerabilities.

Tool Weaknesses

Each of the aforementioned tools comes from a third party and use a generic approach to application component identification and vulnerability detection. This need for generic coverage leads to false negatives, where certain code pathways or application components may not be discovered by dynamic tools or static tools that incorrectly bypass entry points. While static analysis tools can provide verifiable full-coverage of an application, strategies for identifying possibly malicious sources and vulnerable sinks are only as good as the generic frameworks provided by the vendors. Identifying all vulnerabilities within an application requires in-depth knowledge, intelligence, and tuning that vendor tools cannot provide.

In addition, full classes of vulnerabilities are difficult for generic tools to identify, including the authorization and business logic flaw vulnerabilities. For example, a human understands that a regular consumer account should not be able to make critical, administrative changes within the application. While quickly identified by a manual tester, a testing tool cannot make this deduction without upfront classification that certain components of an application should only be accessible to certain roles.

Finally, security or test edge cases are often ignored by testing tools until new research concentrates on that area. Since the tools are developed with certain classes of vulnerabilities in mind, the unknowns or additional cases that might easily be identified may be ignored.

Tools Summary

Overall, continued used of application security tools for identifying security vulnerabilities is recommended, but additional testing approaches are appropriate to cover additional vulnerability classes, edge cases, and full application component coverage. For this, we turn to traditional assurance testing frameworks within the next section.

Unit-Testing Frameworks

Most available languages and frameworks provide a scaffolding for unit testing. Java Spring and ASP.NET MVC provide mock controllers for functional testing, which is often supplemented with third party libraries. Django also contains a framework for exercising application functionality. The problem frequently encountered by the author is a lack of real implementation of these testing framework, especially for security reasons.

While this section focuses specifically on Java Spring, .Net MVC, and Python Django, other modern frameworks languages use similar techniques to provide QA testers with unit-testing functionality. As we are addressing both unit and integration testing, let's look at each technology and how they approach unit and integration-testing.

Java Spring

The Java Spring Framework provides a number of different functions to test everything from simple units to full integration tests. Spring integration tests, in particular, allow for full rendering of JSP files and Spring servlets for testing complete application functionality. Java frameworks for unit-tests are extremely fast when testing functional validation, but don't always provide enough functionality to uncover security vulnerabilities.

It is evident that the Java ecosystem has a long history of unit-testing, and the Spring framework is no exception. Options exist for unit-testing single functions up to complete integration tests by changing between built-in testing frameworks, mock instances, and test attributes. Given the wide variety of options, it can be difficult for novice programmers to pick a test framework and start.

ASP.NET MVC

C# and ASP.NET MVC provide much of the same unit-testing functionality as Java Spring. Simply choosing a testing framework between Microsoft's internal testing framework and a third-party option can be a difficult decision. Developers must take the time to research the different available options and choose a framework that meets the assurance needs of the current project. Build system, coding pipeline tools, and intended distribution platform all play a role in which framework makes the most sense.

In general, the ASP.NET MVC Testing frameworks work by directly calling mocked components of the MVC controller methods using built-in or 3rd-party frameworks. This limits access to rendered HTML and a full HTTP request or response. Full integration-tests require instantiation of a full application server and can be difficult to create.

Python Django

Django's test framework becomes more of a "take-it-or-leave-it" prospect. It uses the standard python unit-test library and is a hybrid of a unit and integration testing

framework. As such, it provides hooks to auto-create a model database that is used for the tests to avoid conflicts with production data. If integrating tests into Django, 3rd-party options do not factor in and the built-in API provides adequate functionality for performing any unit-test from functional to fully integrated.

Testing Frameworks Summary

Any of the above frameworks and tools can be used to implement security unit-tests, but most testing activities seen in the wild using these techniques focus on functional tests of the application, rather than anything security related. Hooking into the built-in frameworks allows for reflection of application functions and some automation, but isn't always used by developers or assurance engineers when implementing tests. Additionally, frameworks dependent on mocking functions concentrate on a small portion of the intended application, making security unit-testing difficult.

Security Unit-Testing Requirements

Now that we have analyzed the existing landscape of security testing tools and unit-testing frameworks, let's define what is required for a developer to implement a security unit-testing framework. In the author's experience, there are three critical pieces to conducting a security unit-test: a functioning application; a way to maintain authentication state; and a repeatable process for interacting with the application.

Functional Application

The first requirement for a security unit testing framework is that the application is functional and running in a "production-like" state. In other words, a successful security unit test can be equated to a successful integration test. Many security issues only manifest when the full application stack is present. For example, without a fully rendered response, it would be impossible to check for client-side XSS attacks or insufficiencies in HTTP header directives.

Authentication State

The next requirement is the ability for the unit-test to maintain authentication state. There are full classes of vulnerabilities in the OWASP Top 10, including Insecure Direct Object Reference and Missing Function Level Access Control that are directly tied to authentication and authorization functions. Without the ability to maintain an authenticated session, unit-tests for these security issues would fail.

Application Interaction

The final requirement for a security unit-testing framework is the ability to interact with the application in a consistent manner. This goes hand in hand with the requirements for maintaining state and a functional application. In practice, however, different frameworks handle tests in manners that may not allow for consistent uptime during the testing process.

To contrast fulfillment of these requirements, view the different methods for starting up an application for testing in Java Spring, .Net MVC, and Django. Java Spring with the Spring Boot Test framework allows for attribute modifiers that instruct JUnit on proper startup during each test class.

```
23
24 @RunWith(SpringJUnit4ClassRunner.class)
25 @SpringBootTest(classes = { MvcConfig.class,MoneyxApplication.class},
26                 webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
27 public class InjectionTest extends MoneyXTestTemplate {
28     @LocalServerPort
29     private int port;
```

Alternatively, unit-testing an ASP.NET MVC application requires that the unit test framework spins up a separate IISExpress process to host the application.

```
private void StartIIS()
{
    var applicationPath = GetApplicationPath(_applicationName);
    var programFiles = Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles);

    _iisProcess = new Process();
    _iisProcess.StartInfo.FileName = programFiles + @"\IIS Express\iisexpress.exe";
    _iisProcess.StartInfo.Arguments = string.Format("/path:\"{0}\" /port:{1}", applicationPath, _iisPort);
    _iisProcess.Start();
}
```

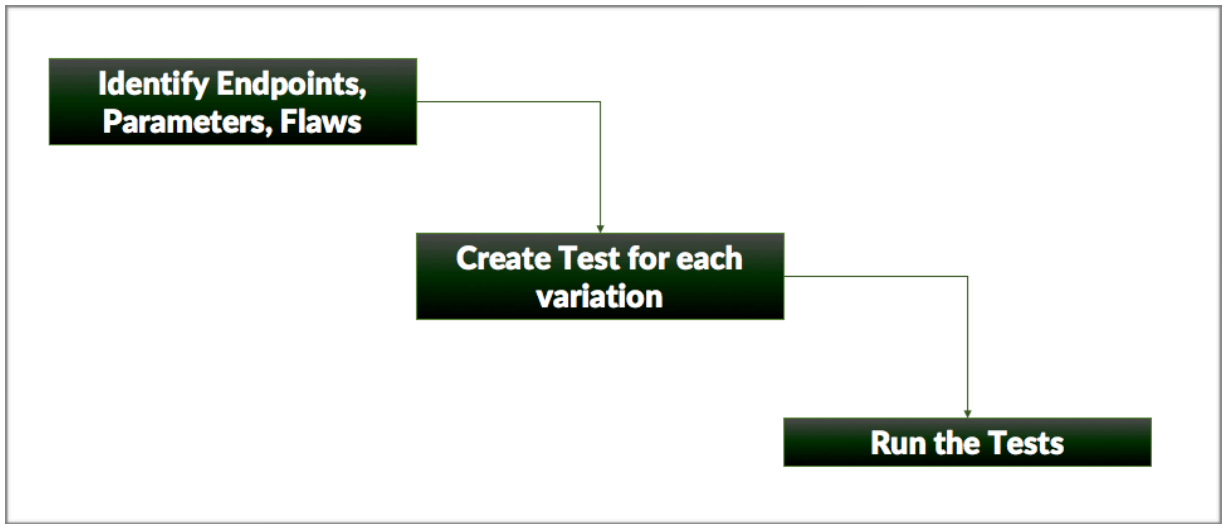
Finally, Django embeds easy access to the full running application through the use of the django.test.Client interface.

```
21 class TestSecurity(TestCase):
22     "Security Tests for Django Secure Coding"
23     fixtures = ['users', 'usersProfiles', 'groups', 'auth_group_permissions', 'taskManagerProjects', 't
24
25     def setUp(self):
26         self.client = Client()
27
28     #def tearDown(self):
29         #Nothing todo yet
30
31     def test_caching(self):
32        "django.nv is vulnerable to sensitive data caching in browsers (hint: birthday)"
33        vulnerable = False
34        req = self.client.login(username='seth', password='soccerlover')
35        res = self.client.get("/taskManager/profile/2")
36        if res.has_header('Cache-Control') and res.has_header('Expires') and res.has_header('Pragma'):
37            if not res['Cache-Control'] == 'no-cache, no-store':
```

Since each approach to presenting a testing surface is different, the approach to unit-testing will vary as well. These differences may not make a huge difference to developers, but assurance and security testers that deal with multiple technologies must take them into account.

Security Unit-Testing Approach

The ability to build a security unit-test does not fully cover an application against all possible vulnerabilities for each function. For full coverage of these flaws, each accessible endpoint and parameter needs to be tested. This means we must build a process or framework to construct the security unit-tests. As with normal assurance testing, a complete suite of security unit-tests is built by following the simple mantra of identify, create, and test.



Identify

Identification within the context of security unit testing is not a trivial task and can take just as long as the actual testing. This stage must identify all applicable vulnerabilities, application endpoints, and available parameters. In a dynamic security test, the scanner performs identification by spidering the application and storing endpoints and parameters for later testing. Static analysis has the advantage of looking at the code for the same values.

For security unit-testing purposes, follow a strategy similar to static analysis tools. By analyzing the code for endpoints and parameters, we can exercise full coverage of the application for each of the different vulnerability types. In addition, manual analysis allows us to target specific endpoints with specific vulnerabilities or payloads that a generic scanner may ignore.

The structure of the application we are testing can ease this identification phase. For example, the Django framework always contains a *urls.py* file that specifies the available URLs for that application. By deconstructing the included regular expression, we know that the following example specifies the application */download/* URL that also includes a `file_id` parameter in the URL.

This file points us in the right direction for application endpoints in Django files and does include URL embedded parameters. However, further analysis of the Django *views.py* file is necessary to identify GET and POST parameters that are not specified in *urls.py.*

Identification of applicable vulnerabilities can also be a time-consuming task. Depending on the application language and intended use, full classes of applications may or may not be applicable. For instance, web applications can start by testing for the OWASP Top 10, but use of a non-memory managed language like C/C++ forces the inclusion of testing memory-management vulnerabilities like buffer overflows.

Create

Once a full list of application parameters, endpoints, and vulnerabilities are identified, it is time to create unit-tests for each one. Keep in mind that a good unit-test is a simple unit-test. The more complicated the test is, the harder it is to know what passed or failed. Instead of writing one test to cover all instances of a vulnerability in the application, each test is treated as a simple bot and will only cover one vulnerability in one parameter on one endpoint. In addition to this, limit each unit-test to a single `Assert` statement in order to avoid confusion.

Test

It is finally time to run the security unit-tests against an application. At this point, timing of tests becomes important, and the use of parallel execution or multiple application endpoints can reduce the overall time of a test.

Security Payload Unit Testing Repository/Runner

In creating security unit-tests, a number of problems are evident when reviewing the current landscape that specifically apply to building effective security tests. Testing payloads, unit-test generation, and endpoint identification are all areas that can take large amounts of time for both existing and new applications. The Security Payload Unit-Testing Repository/Runner (SPUTR) attempts to rectify these issues by introducing limited payloads, a tool for generating and running unit-tests, and parsing of known development frameworks for endpoints.

Testing Payloads

The initial inspiration for SPUTR came while developing security unit-tests for an intentionally-vulnerable application. When attempting to identify and test vulnerable endpoints, certain characteristics were discovered. First, that traditional security payloads are focused on exploitation. Next, the payloads are often redundant. Finally, positive identification of vulnerabilities when using these payloads is not always obvious.

The current set of security payloads, such as fuzzdb, concentrate on successful exploitation of vulnerabilities in order to provide false positive reduction. This behavior is an extension of using generic tests to identify flaws. Without knowledge of application behavior and payload use, the only way to positively identify a vulnerability is to exploit it. Because this is the main goal of the fuzzing lists, list builders concentrate more on application output than input. In building a simple unit-test, however, we concentrate more on the input and uncovering the edge case that are indications of insecure coding practices.

It only follows that such payload lists will contain redundant exploits, as each payload is built to exploit a vulnerability in a different manner. Take an XSS payload as example, the payloads `"><script>alert(123)</script>` and `"%20onmouseover=alert(123)%20` exploit XSS in the exact same way and point to flaws in both input validation and output encoding. An objective look at both payloads shows that running both is effective in identifying multiple possible XSS exploits, but is redundant to accomplishing our goal of identifying possible secure coding flaws. A simple payload of a double quote (") would uncover possible locations in the application output with less overhead.

Differentiation between test payloads and normal application output is critical for identifying security flaws. Without proper tagging of payload input, it is possible that a flaw can be overlooked or falsely attributed to the test input. Only by manipulating input payloads can we be certain that payloads match specific unit-tests. It is easy to see how this would be problematic in the above XSS example. Attempting to identify if a double quote in application output comes from a security test or normal application behavior is close to impossible. However, if we append the random string '`4ab9d`' before and after the double quote, it makes identification of the output programmatically simple.

To alleviate payload issues in relevant vulnerabilities, SPUTR comes with simple payload lists that identify possible escape strings that lead to exploits. For example, double quotes, single quotes, a space, and other observed successful escape strings for XSS are used in generating payloads. Additionally, random identification strings are generated and attached to the payloads upon request. The intended use is for each payload is only used once, but depending on the unit-test may be used multiple times.

Payloads are built off of a manual list of characters maintained in the exploits directory of the SPUTR project. While you can include all possible dangerous characters, they are sorted by vulnerability for further refinement. The `payload_generator` class file contains the following definitions.

```
12  class Payloads():
13
14      chars_dir = os.getcwd() + "/exploit_chars"
15      payloads_dir = os.getcwd() + "/payloads"
16
17      def process_chars_dir(dir):
18          dirs = os.listdir(dir)
19          payloads = []
20          chars = {}
21          for f in dirs:
22              p = os.path.join(dir,f)
23              if os.path.isdir(p):
24                  process_chars_dir(p)
25              else:
26                  file = open(p)
27                  for l in file:
28                      c = l.rstrip('\n')
29                      chars[c] = 1
30          return chars
31
32      #process_chars_dir(chars_dir,"/")
33
34      def generate_payloads(type):
35          #type can be xss, sqli, xml
36          print('Generating payload list for ' + type)
37          dir = os.getcwd() + "/exploit_chars" + "/" + type
38          s = string.ascii_lowercase + string.digits
39
40          payloads = []
41          chars = Payloads.process_chars_dir(dir)
42          for c in chars:
43              r = ''.join(random.sample(s,5))
44              p = r + c + r
45              payloads.append(p)
46          return payloads
```

By tracking unique characters, we further limit the amount of payloads generated and sent to application endpoints while increasing the effectiveness of the security unit-tests. As seen in the following screenshot, the payload values change for each test, but use random strings so that payloads can be tracked through the application.

```
XSS Test for http://localhost:8000/taskManager/search/
=> Payload abtzh&abtzh not filtered for parameter q
=> Payload xfobc<xfobc not filtered for parameter q
=> Payload oi2pj>oi2pj not filtered for parameter q
=> Payload m5thc"m5thc not filtered for parameter q
=> Payload p6zt7'p6zt7 not filtered for parameter q
=> 0/5 passed/total
```

Generating and Running Security Unit-Tests

The next problem that SPUTR alleviates is the generation of security unit-tests. While not intended, security is often an afterthought during application development and leads to extra cost and effort to implement security into the SDLC. This fact is in part responsible for the dependency of security and development on generic security testing tools. By providing a framework and tool for generating and running application-custom unit-tests, identification and remediation of security flaws can be accomplished quickly.

To generate and run a security unit-test, SPUTR uses a JavaScript Object Notation (JSON) configuration file that specifies application endpoints and specific unit-tests to run. The following example shows the use of an example configuration for django.nV that tests for possible SQL Injection, XSS, and Access Control flaws.

```
$ python3 sputr.py --config examples/django.nv-config.json —test
```

```
running sqli tests
Generating payload list for injection/sql
SQL Injection Test for http://localhost:8000/taskManager/search/
=> 13/13 passed/total
running xss tests
Generating payload list for xss
XSS Test for http://localhost:8000/taskManager/search/
=> Payload zhuja&zhuja not filtered for parameter q
=> Payload 5omkz<5omkz not filtered for parameter q
=> Payload c1yjn>c1yjn not filtered for parameter q
=> Payload w54vf"w54vf not filtered for parameter q
=> Payload c6hok'c6hok not filtered for parameter q
=> 0/5 passed/total
running access control tests
Access Control Test for http://localhost:8000/taskManager/search/
=> 1/1 passed/total
running sqli tests
Generating payload list for injection/sql
SQL Injection Test for http://localhost:8000/taskManager/login
=> 26/26 passed/total
```

Notice how the `q` parameter on the `/search/` page of the application is tested for all three flaws, which included 13 injection tests, 5 XSS tests, and 1 access control test. The page does output encode or filter the provided input and fails the XSS tests accordingly.

Parsing Known Development Frameworks

The final security unit-testing problem that SPUTR addresses is automation of the task to parse out endpoints and parameters from the different development frameworks. Support for different frameworks is currently limited to the test frameworks used by the authors in developing SPUTR and will be expanded as new applications and technologies requiring support are identified.

Each import task starts by utilizing route definitions, but requires manual review to make sure that full coverage of all endpoints and parameters is included in the tests. The initial parsing is highly dependent on regular expressions and application reflection, so the addition of new technologies is not a trivial task.

One thing to remember is that SPUTR does not fully eliminate manual review of the unit tests for appropriateness. In it's current form (v1.0), it requires manual intervention for defining appropriate vulnerability checks and parameters for each identified endpoint.

The following example shows how SPUTR is used to generate a configuration file for a Django application, including some additional python requirements.

- Step 1: Install Django Extensions
  - `pip install django-extensions`

- Step 2: Run the Show URL Extension and dump out the routed URLs.
  - `python manage.py show_urls >> urls.py`

```
mb36:django.nV slaw$ python3 manage.py show_urls
/       taskManager.views.index index   login_required
/favicon.ico/   taskManager.urls.<lambda>
/taskManager/   taskManager.views.index taskManager:index
/taskManager/<project_id>/<task_id>/    taskManager.views.task_details  taskManager:task_details
/taskManager/<project_id>/<task_id>/note_create/        taskManager.views.note_create   taskManager:note_create
/taskManager/<project_id>/<task_id>/note_delete/<note_id>       taskManager.views.note_delete   taskManager:note_delete
/taskManager/<project_id>/<task_id>/note_edit/<note_id> taskManager.views.note_edit     taskManager:note_edit
/taskManager/<project_id>/edit_project/ taskManager.views.project_edit  taskManager:project_edit
/taskManager/<project_id>/manage_tasks/ taskManager.views.manage_tasks  taskManager:manage_tasks
/taskManager/<project_id>/project_delete/       taskManager.views.project_delete        taskManager:project_delete
/taskManager/<project_id>/project_details/      taskManager.views.project_details       taskManager:project_details
/taskManager/<project_id>/task_complete/<task_id>       taskManager.views.task_complete taskManager:task_complete
/taskManager/<project_id>/task_create/  taskManager.views.task_create   taskManager:task_create
/taskManager/<project_id>/task_delete/<task_id> taskManager.views.task_delete   taskManager:task_delete
/taskManager/<project_id>/task_edit/<task_id>   taskManager.views.task_edit     taskManager:task_edit
/taskManager/<project_id>/upload/       taskManager.views.upload        taskManager:upload
/taskManager/change_password/   taskManager.views.change_password       taskManager:change_password
/taskManager/dashboard/ taskManager.views.dashboard     taskManager:dashboard
/taskManager/download/<file_id>/        taskManager.views.download      taskManager:download
/taskManager/downloadprofilepic/<user_id>/      taskManager.views.download_profile_pic  taskManager:download_profile_pic
```

```
$ python3 sputr.py --generate --apptype django --output ../
test.config --appdir ../../nVisium/Code/django.nV
```

```
{
    "token": {
        "name": "cookie_name",
        "value": "cookie_value"
    },
    "creds": {
        "username": {
            "name": "username",
            "value": "testuser"
        },
        "password": {
            "name": "password",
            "password": "temppass"
        }
    },
    "csrf": {
        "pattern": "^regexpattern$",
        "name": "csrftokenname"
    },
    "domain": {
        "host": "localhost:8000",
        "protocol": "http://",
        "login_url": "http://localhost:8000/taskManager/login",
        "auth_url": "http://localhost:8000/taskManager/dashboard"
    },
```

Remember, the above configuration file must be updated with proper CSRF patterns, user credentials and cookie names and values. In addition, review the endpoints for appropriateness and add parameters and tests for each.

Planning the Future with SPUTR

SPUTR is in its infancy of usefulness, with big plans for the future. As mentioned previously, support for additional languages and frameworks for test generation will continue as long as the application is in development. On top of this, the next version of the tool will feature expanded support for additional vulnerabilities, including the remainder of the OWASP Top 10 and other classes that make sense. The last big push will be related to speed of testing with SPUTR, as it is currently single threaded. Since the tool is meant to be used within a CI pipeline, multithreading and other speed improvements will only enhance its effectiveness.

Conclusion

Even though the development industry is still lacking when to comes to security testing, there is hope. Use of existing security tools within the software pipeline has increased an organization's ability to find and fix security vulnerabilities, but custom tests will always perform more efficiently than their generic equivalent. By implementing custom security unit and integration-tests, an organization can increase their assurance that security flaws do not exist in critical code bases. Multiple frameworks exist for unit and integration testing, and SPUTR is a new tool that can be utilized for this purpose.

References

Java Spring Framework Integration Testing - http://docs.spring.io/spring/docs/current/spring-framework-reference/html/integration-testing.html
Java Spring Boot Testing - http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html
Writing Unit Tests for the .NET Framework with the Microsoft Unit Test Framework for Managed Code - https://msdn.microsoft.com/en-us/library/hh598960.aspx
Unit Test Basics - https://msdn.microsoft.com/en-us/library/hh694602.aspx
Unit Testing in ASP.NET MVC Applications - https://msdn.microsoft.com/en-us/library/ff936235(v=vs.100).aspx
FuzzDB Project - https://github.com/fuzzdb-project/fuzzdb
OWASP Testing Project - http://www.owasp.org/index.php/OWASP_Testing_Project