The Report Committee for Jason Andrew Longoria
Certifies that this is the approved version of the following report:

Prioritizing security regression test cases using threat models

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Sarfraz Khurshid

Suzanne Barber

Prioritizing security regression test cases using threat models

by

Jason Andrew Longoria, B.S.Ch.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2016

Abstract

Prioritizing security regression test cases using threat models

Jason Andrew Longoria, M.S.E.

The University of Texas at Austin, 2016

Supervisor: Sarfraz Khurshid

When existing software is modified, regression testing provides an approach to gain confidence that no unexpected security vulnerabilities have been introduced. If faults or vulnerabilities were introduced by the change, it is beneficial to identify them as soon as possible. Prioritizing regression test cases by their risk exposure improves the likelihood that faults will be found early. This paper reviews regression test case prioritization methods and provides an example prioritization of security regression test cases based on a threat model.

Table of Contents

List of Tables

List of Figures

Introduction

Industrial applications use software to run critical processes, with risks to life and property if those processes fail. As the number of human operators decreases, each operator is tasked with a larger scope and software-based automation is deployed to drive productivity. At the same time, network connectivity and remote access are becoming increasingly expected. This creates an environment where threats to industrial processes via software are complex and diverse. Thorough software testing is necessary to identify weaknesses so that they can be addressed before the software is deployed to a live process.

Software testing is broadly categorized into functional testing and security testing. Functional testing is focused on assuring that all required features have been correctly implemented from the perspective of a user using the software as intended. Security testing is focused on identifying unwanted functionality that could be exploited by a user using the software in ways that are not intended.

Software is never truly finished. There are inevitably modifications to introduce new functionality, correct flaws, or maintain compatibility in a changing environment. With each change comes the possibility of unintended side effects, or regressions. Regression testing provides confidence that the previously-working parts of the software continue to function as intended, and aims to identify any newly introduced flaws that should be addressed [1]. Both functional regression tests and security regression tests are run to identify new flaws introduced by a change.

It is usually impractical due to time and resource constraints to retest an entire software product every time a change is made. Therefore it is necessary to identify which test cases are most important for regression testing. Test case selection [3, 4, 5] chooses a

subset of test cases to run, and test case prioritization [6, 7, 8] identifies an ordering that is likely to uncover flaws early in the testing process. Both strategies are most effective when they are applied in a systematic way.

A systematic technique for analyzing software security is threat modeling. A threat model [12, 13] is developed during the software design process, refined during development, and modified when the software is later changed. The purpose of a threat model is to highlight assets that should be protected, threats to those assets, and mitigations of those threats, in a format that is easily understood by the various stakeholders in a software project.

In this paper I will explore methods for selecting and prioritizing test cases for security regression testing. I will start with a review of literature relevant to test case selection and prioritization followed by literature relevant to the use of threat models. I will then demonstrate with a hypothetical example how a threat model can be applied for test case prioritization. Finally, I will discuss open questions that suggest opportunities for future work.

Literature Review

TEST CASE SELECTION AND PRIORITIZATION

Three broad categories of test management are test suite minimization, test case selection, and test case prioritization. Test suite minimization is an ongoing maintenance activity that is not specific to any particular code change [4]. As code is developed and new test cases are created, minimization prevents the size of the test suite from becoming unmanageable by pruning redundant and obsolete cases. Test case selection focuses on a specific code change and chooses tests from the test suite that are expected to be most effective at uncovering faults caused by that change [3, 4, 5]. Acknowledging that schedule and resource limitations might make full execution of an entire test suite infeasible, test case prioritization determines an order for running tests so that faults are likely to be uncovered as early as possible [6, 7, 8]. Even when the whole test suite could be run, uncovering a fault quickly allows a fix to be implemented and retested promptly.

In 1990, Leung and White described a method for test selection during integration testing [2]. They proposed a "firewall" for integration-level regression testing. After unit tests have been run to validate changes in individual modules, pairwise interactions of modules are tested, based on a call graph that illustrates all the interactions between modules. Their work demonstrated that it is only necessary to test interactions with modules that directly interact with the changed modules. With well-designed test cases, any regressions in more distant areas of the program should be uncovered when testing direct interactions with the changed components. In this way the modules immediately adjacent to the revised modules provide a firewall, beyond which no further regression testing is required, reducing the number of tests that need to be run following a software revision.

Testing occurs at several phases in the software development process. Unit testing focuses on the functionality of individual components. Integration testing combines components and tests their interactions. Function testing compares the completed software to its functional specification. System testing is performed on the whole system of hardware and software, broadly evaluating nonfunctional requirements such as system performance in addition to functional requirements. In 1992, Leung and White extended their prior work to apply to function testing, which is often done in a black-box manner without access to the code [3]. Instead of a call graph based on implementation details, they used a data flow graph based on specifications. The same firewall concept from their prior work was applied to functional testing to minimize the number of functional tests required following a software change.

In 1996, Rothermel and Harrold conducted a literature review to explore criteria that can be used to judge the effectiveness of different techniques for regression test case selection [4]. One dimension of the problem is the cost of each regression test versus the effectiveness of each test at uncovering faults. There is a tradeoff between spending resources to carefully evaluate which tests to run versus being less selective and running more tests. The relative cost of each approach is system-dependent. A system in which tests are expensive to run will benefit from running fewer tests, while a system in which faults are costly will require greater inclusiveness. They also considered generality -- whether the test selection technique is language-specific (intraprocedural) or code-agnostic (interprocedural). Intraprocedural techniques are more readily automatable, while interprocedural techniques are better suited to black-box testing. Their analysis presented a groundwork and vocabulary for evaluating regression test selection techniques.

Chen, et al. in 2002 described a method of test case selection based on risk exposure [5]. They built an activity diagram based on the software specification, along with a

traceability matrix to associate model elements with test cases. They then demonstrated that changes either to the model or to the software can guide test case selection by focusing on the portions of the model affected by the change. Risk exposure was calculated for each test case based on the importance of the tested modules to the customer and to the functionality of the program. Historical data of faults uncovered by each test case were then used to estimate the likelihood that a test case will uncover faults, and the severity of those faults. Based on these inputs, each test case was assigned a numerical risk exposure value. Test cases were selected until a predetermined target coverage level was attained. While that study focused on test case selection, the same method could be used for test case prioritization by continuing through the whole test suite.

In 2008, Ma and Zhao proposed a similar technique for test case prioritization rather than selection [6]. They assigned quantitative values to each module in the software based on importance of the module and fault proneness. They then used a module call graph to identify the modules traversed by each test case. By first running the test cases that cover the highest-risk modules, this method provided a mechanism for prioritizing the test cases that was found to be effective at improving the rate of fault detection.

In 2005, Srikanth, et al. described a method of test case prioritization based on requirements rather than internal program structure [7]. They assigned quantitative values to each requirement based on its value to the customer, complexity of implementation, volatility, and fault proneness. Test cases were be prioritized based on which requirements they test. When applying this method to two example programs they found that the method was effective at improving the rate of fault detection.

In 2000, Elbaum, et. al. compared different techniques for test case prioritization [8]. Using rate of fault detection as the criterion for evaluating various methods, they found that prioritization techniques can be effective at different levels of abstraction -- both in

integration testing and in function testing. This is useful since higher levels of abstraction are less intrusive to implement, requiring no instrumentation in the code itself.

While much of the literature is based on techniques where source code is available, test teams in practice may not have complete access to the code or may lack the expertise to analyze the code in depth. Even where source code is available, there are advantages in having tests performed by an independent team that is not biased by knowledge of behind-the-scenes implementation details. For this reason, an active area of research is black-box test case prioritization. Caliebe, et. al. described a test case prioritization technique for automotive testing [9]. Automotive applications integrate software components from many different development teams, making in-depth code analysis of a finished product impractical. Based on system architecture and requirements, without considering implementation details in the source code, they developed a Component-Dependency Model (CDM). They then used the model to identify which components would be most affected by a change, thus providing prioritization for version-specific regression testing. This is similar in concept to the test selection method developed by Leung, et. al., but at a higher level of abstraction that does not depend on implementation details.

THREAT MODELING

In 2002, Microsoft CEO Bill Gates issued a memo highlighting the importance of "Trustworthy Computing" [10]. At a time when web-based applications were rapidly being developed, his argument was that customers need to have trust before they will embrace new technologies. Rather than thinking of security as an afterthought, it should be integrated into the design process from the beginning and given top priority. As part of this effort to refocus on security, Microsoft introduced the STRIDE method of classifying risks into six buckets: Spoofing, Tampering, Repudiation, Information disclosure, Denial of

service, and Elevation of privilege. This facilitated systematic consideration of security threats in a software product. They also released their Threat Modeling Tool, which is still widely used for producing Data Flow Diagrams and documenting risks and mitigations.

In 2005, Sindre and Opdahl suggested a method for extending existing use-case analysis to include misuse-case analysis [11]. This was in response to the observation that stakeholders developing software requirements were comprehensive when describing how the software should behave, but frequently left gaps with regard to how the software should not behave. The intention of misuse case analysis was to spark discussion of security requirements early in the development process. While the misuse cases are not a complete threat model by themselves, they can serve as a starting point for discussing vulnerabilities in an application.

Lindqvist, et. al. described a method for using Correlated Attack Modelling Language (CAML) to facilitate automatic detection of attack scenarios during the software design process [12]. Individual attack steps may be common to different attack sequences; one example is a vulnerability in IIS that might be present in any web application lacking a particular patch. By providing a standardized vocabulary of these steps, CAML enables security analysts to focus on complete multi-step attack sequences, represented as attack trees. Each node in the tree represents a specific vulnerability. A composite attack traverses the tree to use multiple vulnerabilities in sequence. A common combination of attack sequences can be grouped into an “attack pattern”, analogous to the concept of a “design pattern” in software development. This grouping allows higher-level modeling to be conducted.

Marback, et. al. elaborated on the use of threat trees to generate test cases [13]. An attack in which an adversary executes arbitrary commands in a system can be decomposed into the tasks of gaining access to the system, attaining elevated privilege, and issuing

commands. Each of these tasks can be further decomposed into subtasks and alternative methods: gaining access can be achieved by spoofing identity, by brute-force password attack, or by social engineering. The various alternatives (related to each other as "OR") and sequences (related to each other as "AND") form a tree. Following the nodes in a tree from leaf to root describes a theoretical attack strategy that might be employed. Test cases were automatically generated for each attack strategy, with the goal of determining whether or not each strategy is effective on the system in practice.

An alternative threat modeling methodology is Trike, described by Saitta, et. al. in 2005 [14]. It seeks to enumerate all threats inherent in the business logic of the application, rather than modeling attacks from an attacker's perspective. It uses nested Data Flow Diagrams to depict the interactions among actors, assets, actions, and rules at varying levels of detail. Rather than using the six categories of STRIDE, Trike considers every action to be one of four options: Create, Read, Update, or Delete. An asset-action interaction matrix is generated to summarize whether each combination of actor and action is allowed, disallowed, or allowed with rules. Filling in the complete interaction matrix is one way that the analysis is systematic rather than ad hoc.

Approach

MOTIVATING EXAMPLE

To demonstrate the use of a threat model for regression test case prioritization, consider this example from the domain of industrial process control.

An operator at a refinery logs into the human-machine interface (HMI) of a distributed control system. After being authenticated, the operator can view the current operating parameters of the process and can change a controller's setpoint by clicking up and down buttons on an HMI display. Whenever the operator makes a setpoint change, the change is communicated to the controller and is also recorded in a change management database. The change management database stores the operator's name, the old and new setpoints, and the current time, which is synchronized across various machines in the control system by periodic updates from a time server. An engineer can log in to view the history of setpoint changes from the change management database but cannot modify any records and cannot make any changes that affect the process.

THREAT MODEL

The first step in creating the threat model is to decompose the application into its component parts. Systematically considering every entry point, asset, and trust level will provide a basis for evaluating threats. This is typically done at multiple levels of abstraction: once an asset is identified, it can be further decomposed into its component parts. For brevity in this example, only a single level of abstraction is considered.

Focusing on the HMI as the subject of this analysis, entry points summarized in Table 1. These are avenues that an attacker could use to gain access to the HMI. Each entry point is designed to allow access only for entities with an appropriate trust level.

Entry Point	Description	Trust level
Control network	HMI receives process data from control network.	Process
Control network	HMI sends setpoint command to controller	Process
Operator display	HMI displays process data to Operator	Operator
Operator display	HMI receives commands from user	Operator
Database	HMI sends setpoint changes to Database (one way).	Database

Table 1: Entry points.

Trust levels are summarized in Table 2. These are the users and external machines that communicate with the HMI. Each entity interacting with the HMI undergoes authentication to attain the appropriate trust level.

Trust Level	Description
Operator	A logged-in user who can view current process data and issue commands via the HMI
Administrator	A logged-in user who can create users and modify their permissions
Engineer	A logged-in user who can view historical records from the change management database
Database	A trusted database that stores history of setpoint changes
Process	Instruments and final control elements that provide runtime process data and receive runtime process commands

Table 2: Trust levels.

Assets are the data entities that have value and could be attack targets. The assets in this example are summarized in Table 3.

Asset	Description	Trust Level
Process data	Current operating parameters for the industrial process	Operator
Process commands	Commands issued to controllers that affect the industrial process	Operator
Change history	Historical record of setpoint changes	Engineer
User data	User login credentials and authorization levels	Administrator

Table 3: Assets.

Once the component parts of the system have been enumerated, a data flow diagram (DFD) can be constructed. The DFD shows each entity and asset in the system, with arrows drawn to show relationships between them. Trust levels are represented as boundaries between elements. The DFD for this example is shown in Figure 1.

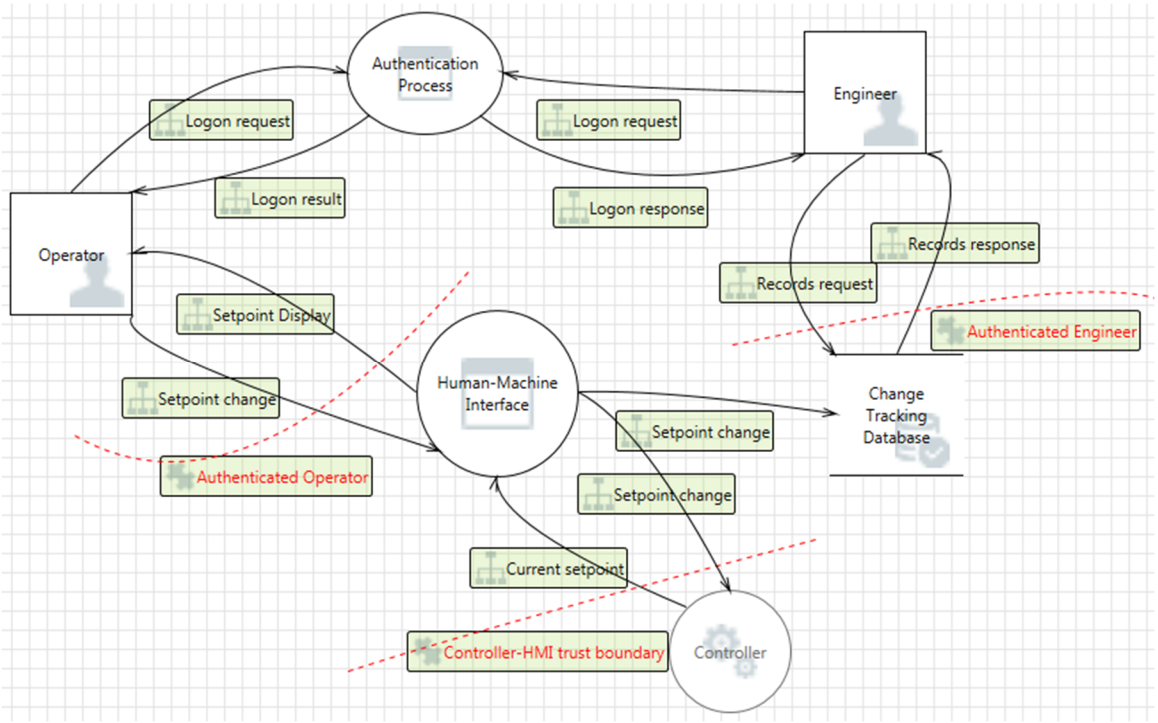

Figure 1: Data Flow Diagram of an example threat model

To generate a list of possible threats using the STRIDE model, each entity and relationship in the model is systematically evaluated for Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. The Microsoft Threat Modeling Tool provides a starting point for this analysis by generating possible threats based on the Data Flow Diagram. For the diagram in Figure 1, 54 potential threats were generated. A sample of those threats to be considered in this example is shown in Table 4.

ID	Category	Description	Severity
Threat001	Spoofing	Operator may be spoofed by an attacker and this may lead to unauthorized access to Human-Machine Interface.	High
Threat002	Elevation of Privilege	Human-Machine Interface may be able to impersonate the context of Operator in order to gain additional privilege.	High
Threat003	Spoofing	Change Tracking Database may be spoofed by an attacker and this may lead to data being written to the attacker's target.	Low
Threat004	Tampering	Potential SQL Injection Vulnerability for Change Tracking Database	Medium
Threat005	Denial of Service	Does Human-Machine Interface or Change Tracking Database take explicit steps to control resource consumption?	Medium
Threat006	Spoofing	Human-Machine Interface may be spoofed by an attacker and this may lead to unauthorized access to Controller.	High
Threat007	Spoofing	Controller may be spoofed by an attacker and this may lead to data being written to the attacker's target.	Low
Threat008	Spoofing	Controller may be spoofed by an attacker and this may lead to inaccurate display of current setpoint on the HMI.	High
Threat090	Tampering	Data flowing from HMI to Database may be tampered with by an attacker. This may lead to corruption of Change Tracking Database.	Medium
Threat010	Information Disclosure	Improper data protection of Change Tracking Database can allow an attacker to read information not intended for disclosure.	Low

Table 4: Sample of threats generated by Microsoft Threat Modeling tool.

After potential threats have been identified, the importance of each threat is evaluated and mitigations to those threats are considered. Threats with low severity might be accepted with no mitigation. Other threats may be mitigated by security controls such as authentication, cryptography, or logging.

TEST CASES

Security testing consists of tests to verify that mitigations work as intended as well as tests to verify that the software does not behave in ways that are unintended. To verify that the identified mitigations are adequate, test cases attempt to exploit the threats. Additional test cases are developed to explore vulnerabilities that were not mitigated. Table 5 contains a sample of test cases for this example. Each test case tests a specific portion of the system.

ID	Title
Test010	HMI does not display current setpoint to unauthenticated user
Test020	HMI does not accept setpoint change from unauthenticated user
Test030	HMI does not accept current setpoint from unauthenticated controller
Test040	HMI does not send invalid setpoint command
Test050	HMI does not send setpoint change to unauthenticated destination
Test060	HMI does not consume excessive resources sending setpoint commands
Test070	Database accepts edits only from authenticated HMI
Test080	Database provides records only to authenticated Engineer

Table 5: Sample of test cases.

It is assumed that these test cases were already created and added to the test suite during software development. Furthermore, it is assumed that the test cases were already run and any deficiencies were fixed. Thus, it is expected that the test cases should pass

when used for regression testing. Failures will indicate a problem resulting from the change, either with the system under test or with the test case.

Each threat may be tested by one or more test cases, and each test case may test one or more threats. A matrix can be generated to relate threats to test cases. An X indicates that the threat is tested by the test case

	Test010	Test020	Test030	Test040	Test050	Test060	Test070	Test080
Threat001	X	X						
Threat002		X						
Threat003					X			
Threat004				X				
Threat005						X		
Threat006		X						
Threat007					X			
Threat008			X					
Threat009							X	
Threat010								X

Table 6: Matrix relating test cases to threats.

REGRESSION

Now the example is extended to introduce a change. The HMI is changed so that in addition to clicking up and down buttons to adjust the controller setpoint, the operator will have the ability to enter a numerical setpoint using the keyboard.

A new threat becomes possible when the HMI is modified to allow text entry via the keyboard. If the input is not properly sanitized, invalid text could be entered into the HMI and propagated to the controller or to the change management database. Possible problems include buffer overflows and SQL injection attacks. To account for these new

vulnerabilities, the test suite should be modified to add test cases for these vulnerabilities. The remaining test cases are then run for regression testing. The focus here is on ordering the pre-existing regression tests.

Based on the firewall concept described by Leung and White [2], the test cases most relevant to this change are those that test the HMI itself and those that test the HMI's direct interactions with other elements in the system. More distant relationships need not be prioritized since any regressions should be evident from direct interactions with the HMI. Table 7 lists the interactions that directly involve the HMI. Each of these interactions is covered by test cases enumerated previously in Table 5.

Interaction	**Test cases**
Operator views setpoint from HMI	Test010
Operator modifies setpoint via HMI	Test020
HMI obtains setpoint from controller	Test030
HMI sends setpoint change to controller	Test040, Test050, Test060
HMI sends setpoint change to change management database	Test040, Test050, Test060

Table 7: Interactions directly involving the HMI.

The test cases enumerated in Table 7 are the ones that will be selected for prioritization based on this particular change. Other tests in the suite are more distantly related to this change; based on the firewall concept, any faults should be most readily apparent in the test cases that directly involve the HMI.

Using the method described by Chen [5], risk exposure for each test case is calculated from the combination of cost and severity probability. In this context, cost refers to the severity of the consequences when a requirement tested by a test case is not met. Severity probability measures the number and severity of defects covered by each test case

in historical test runs. Multiplying the cost of each test case by its severity probability yields the risk exposure. Test cases are prioritized in order of decreasing risk exposure, so that the test cases most likely to uncover costly faults are run first.

While Chen, et al. were focused on functional testing based on requirements analysis, their method can be adapted for security testing based on threat analysis. Instead of using an activity diagram to guide functional testing for what the software should do, the data flow diagram showing security interactions is used to guide security testing for what the software should not do.

Risk exposure of each test case is based on the history of faults discovered by each incident. A test case that has been run many times but uncovered few faults has low risk exposure. A test case that has uncovered many faults has high risk exposure. A test case that has not yet been run, or has been run very few times, is assumed to have a moderate risk exposure until more data is gathered. For the purpose of this hypothetical example, suppose test case risk exposures are as shown in Table 7.

ID	**Risk Exposure**
Test010	High
Test020	Low
Test030	Low
Test040	High
Test050	Medium
Test060	High

Table 8: Risk exposure of each test case based on historical test runs.

Severity of each potential threat was previously established during threat model development, as shown in Table 4. To assign these severities to test cases, the highest-

severity threat is selected for each test case. Table 9 starts with the matrix relating test cases to threats from Table 6, selecting only those tests that directly relate to the HMI, and replaces each X with the severity of the associated threat from Table 4. The maximum for each test case is then summarized in the last row.

	Test010	Test020	Test030	Test040	Test050	Test060
Threat001	High	High				
Threat002		High				
Threat003					Low	
Threat004				Med		
Threat005						Med
Threat006		High				
Threat007					Low	
Threat008			High			
Maximum	High	High	High	Med	Low	Med

Table 9: Determining threat severity for each test case.

Table 10 combines the threat severity from Table 9 with the risk exposure from Table 8 for each test case.

ID	**Risk Exposure**	**Severity**
Test010	High	High
Test020	Low	High
Test030	Low	High
Test040	High	Medium
Test050	Medium	Low
Test060	High	Medium

Table 10: Threat severity and risk exposure for each test case.

To determine a ranking for the test cases it is necessary to represent qualitative values “high”, “medium” and “low” as numerical values. The numerical value is not an inherently meaningful quantity since the assignment of numbers to qualitative values is subjective. Nevertheless, the numbers are useful for comparing test cases to one another for the purpose of prioritization. Table 11 shows the result from assigning values high=5, medium=3 and low=1. Risk exposure is multiplied by severity to determine test case priority.

ID	Risk Exposure	Severity	Product
Test010	5	5	25
Test020	1	5	5
Test030	1	5	5
Test040	5	3	15
Test050	3	1	3
Test060	5	3	15

Table 11: Assigning severity and risk exposure to each test case.

Finally, sorting the test cases results in the prioritization:

ID	Product
Test010	25
Test060	15
Test040	15
Test030	5
Test020	5
Test050	3

Table 12: Prioritized ordering of test cases.

Analysis

Since the risk exposure is calculated based partly on data from historical test runs, this method becomes more useful over time as the same regression test cases are run repeatedly on successive builds. While this example used qualitative values to categorize risk exposure, a more quantitative value could be attained by dividing the number of faults found by each test case by the number of times the test case has been run. Of course, this requires a quick way to summarize the historical runs of each test case, the ease of which depends on the test case management tool used. It also requires the history to be accurate. If testers omit recording some test runs or make mistakes when recording the results it could result in inaccurate risk exposure values.

Automated testing, with results automatically recorded in the test case management tool, is particularly well-suited for historical test run analysis. When automation is used it may be unnecessary to prioritize individual test cases within a suite because the entire suite can be run with little manual intervention. In that scenario this method is could be useful for selecting and prioritizing whole test suites.

In regression testing, the important consideration is the likelihood that a vulnerability was introduced by the change. In the example, the Engineer's access to the database is not related to the method that an Operator uses to enter a setpoint into the HMI. Therefore, test cases that involve the Engineer's access to the database are low priority. On the other hand, test cases that involve changing the setpoint are likely to be impacted if the method of setpoint change is changed and therefore have a higher likelihood of vulnerability. These are the test cases that are the focus of risk-based prioritization.

Threat models are only useful when they are accurate. Maintaining and modifying threat models whenever the software changes is an organizational challenge. Much of the

existing research focuses on technical aspects of threat models. It would be useful to study the processes and procedures that successful organizations use to keep their models up-to-date.

This example used qualitative categories for threat severity. Refining those values to determine a more quantitative threat severity of each test case is a possible area for further exploration. Whether or not it is worthwhile to spend resources evaluating risk more precisely would depend on the system being tested: a system in which tests are expensive to run will derive the most benefit from careful prioritization. It would be interesting to research various methods for assigning numerical values to threat severity, and the tradeoffs in actual implementation of those methods.

Conclusions

Risk-based models of test case selection can assist a test engineer in finding faults as soon as possible. When time and budget are constrained it may not be feasible to execute the entire suit of test cases. Test case prioritization is a strategy for running the tests with the highest likelihood of finding a severe fault early. Previous research in model-based regression testing has largely focused on functional testing. Similar methods can be used for security testing by incorporating threat severities into the priority calculation. This depends on developing a threat model of the system and keeping the model up-to-date as the software changes.

References

[1] Ammann, P., & Offutt, J. (2008). Introduction to software testing. Cambridge University Press.

[2] Leung, H. K., & White, L. (1990, November). A study of integration testing and software regression at the integration level. In Software Maintenance, 1990, Proceedings., Conference on (pp. 290-301). IEEE.

[3] White, L. J., & Leung, H. K. (1992, November). A firewall concept for both control-flow and data-flow in regression integration testing. In Software Maintenance, 1992. Proceerdings., Conference on (pp. 262-271). IEEE.

[4] Rothermel, G., & Harrold, M. J. (1996). Analyzing regression test selection techniques. IEEE Transactions on software engineering, 22(8), 529-551.

[5] Chen, Y., Probert, R. L., & Sims, D. P. (2002, September). Specification-based regression test selection with risk analysis. In Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research (p. 1). IBM Press.

[6] Ma, Z., & Zhao, J. (2008, December). Test case prioritization based on analysis of program structure. In 2008 15th Asia-Pacific Software Engineering Conference (pp. 471-478). IEEE.

[7] Srikanth, H., Williams, L., & Osborne, J. (2005, November). System test case prioritization of new and regression test cases. In 2005 International Symposium on Empirical Software Engineering, 2005. (pp. 10-pp). IEEE.

[8] Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2000). Prioritizing test cases for regression testing (Vol. 25, No. 5, pp. 102-112). ACM.

[9] Caliebe, P., Herpel, T., & German, R. (2012, April). Dependency-based test case selection and prioritization in embedded systems. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (pp. 731-735). IEEE.

[10] Gates, B. (2002). Trustworthy computing. Microsoft internal memo. <https://news.microsoft.com/2012/01/11/memo-from-bill-gates>

[11] Sindre, G., & Opdahl, A. L. (2005). Eliciting security requirements with misuse cases. Requirements engineering, 10(1), 34-44.

[12] Cheung, S., Lindqvist, U., & Fong, M. W. (2003, April). Modeling multistep cyber attacks for scenario recognition. In DARPA information survivability conference and exposition, 2003. Proceedings (Vol. 1, pp. 284-292). IEEE.

[13] Marback, A., Do, H., He, K., Kondamarri, S., & Xu, D. (2013). A threat model-based approach to security testing. Software: Practice and Experience, 43(2), 241-258.

[14] Saitta, P., Larcom, B., & Eddington, M. (2005). Trike v. 1 methodology document <http://dymaxion. org/trike/Trike_v1_Methodology_Document>