Leveraging SecDevOps to Tackle the Technical Debt Associated with Cybersecurity Attack Tactics

Clemente Izurieta
Montana State University
Bozeman, MT, USA
clemente.izurieta@montana.edu

Mary Prouty
Georgia Institute of Technology
Atlanta, GA, USA
meprouty@gatech.edu

ABSTRACT

Context: Managing technical debt (TD) associated with external cybersecurity attacks on an organization can significantly improve decisions made when prioritizing which security weaknesses require attention. Whilst source code vulnerabilities can be found using static analysis techniques, malicious external attacks expose the vulnerabilities of a system at runtime and can sometimes remain hidden for long periods of time. By mapping malicious attack tactics to the consequences of weaknesses (i.e. exploitable source code vulnerabilities) we can begin to understand and prioritize the refactoring of the source code vulnerabilities that cause the greatest amount of technical debt on a system. Goal: To establish an approach that maps common external attack tactics to system weaknesses. The consequences of a weakness associated with a specific attack technique can then be used to determine the technical debt principal of said violation; which can be measured in terms of loss of business rather than source code maintenance. Method: We present a position study that uses Jaccard similarity scoring to examine how 11 malicious attack tactics can relate to Common Weakness Enumerations (CWEs). Results: We conduct a study to simulate attacks, and generate dependency graphs between external attacks and the technical consequences associated with CWEs. Conclusion: The mapping of cyber security attacks to weaknesses allows operational staff (SecDevOps) to focus on deploying appropriate countermeasures and allows developers to focus on refactoring the vulnerabilities with the greatest potential for technical debt.

CCS CONCEPTS

• General and Reference Surveys and overviews • Software and its engineering

KEYWORDS

quality assurance, software quality, technical debt; cybersecurity

1 INTRODUCTION

Various techniques have been used to quantify Technical Debt (TD); however, none have specifically focused on measuring the potential TD of live security attacks that affect systems. The growing number of cybersecurity attacks and their frequency are forcing organizations to pay significantly more attention to security threats. To address cyber-attacks, organizations (including federal government departments) are starting to rely on a SecDevOps [1] approach where *operations* (Ops) focuses on deploying countermeasures (manual and automatic) and *developers* (Dev) focus on refactoring those aspects of source code that minimize the technical debt associated with the vulnerabilities revealed by the malicious attacks. SecDevOps *"(also known as DevSecOps and DevOpsSec) is the process of integrating secure development best practices and methodologies into development and deployment processes which DevOps makes possible."*[1]

Many tools exist that provide metrics-based analysis in terms of the number of vulnerabilities found in a system; however, these tools are executed by developers independently of observations made by operations staff; thus, the prioritization of which vulnerabilities to address may be significantly different than if developers had a communication channel to first responders. Furthermore, operations staff are the first to effect countermeasures from live cyber-attacks. Using a SecDevOps approach, this information can be made available to developers immediately. The consequences of said attacks can be weighed against each other in terms of the technical debt affecting software maintainability but more importantly, in terms of the consequences to the business if a vulnerability is successfully exploited. *"Repairing the damage can be very costly. The TD interest associated with such a weakness can grow significantly at the moment an attacker is successful."* [2]

Enumerations of rules have been established by the greater community (i.e., CVE [2], CWE [3], and CERT [4]) to explore vulnerabilities and weaknesses from different perspectives. These are most valuable to developers, not to operations staff. Mitre®'s Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK)[5] framework is a knowledge base of adversary tactics and techniques based on community contributions from real world observations. It provides a perspective from the attacker's point of view and focuses on describing the tactics and techniques employed in post compromise scenarios. Tactics are subdivided into multiple techniques that describe specific ways in which an adversary can try to achieve a goal. This perspective is most useful to operations staff.

Izurieta et al. [2] is working on ways to operationalize ISO [3][4] standards using Quamoco [5][7] and QATCH [6] to include the assessment of technical debt principal associated with security weaknesses in more intuitive ways than by just providing

[1] https://blog.sqreen.io/secdevops/
[2] https://cve.mitre.org/
[3] https://cwe.mitre.org/
[4] https://www.sei.cmu.edu/about/divisions/cert/index.cfm
[5] https://attack.mitre.org/

vulnerability counts[6]. In this position study, we propose extending this approach further by first mapping the techniques and tactics encountered by Ops from the ATT&CK framework to the CWE consequences thus linking attacks from Ops to Dev. The effects of this mapping will help developers prioritize the technical debt observed from live attacks to source code that is relevant to the attack. In many cases these attacks are sleeping cells, but their discovery is a valuable asset when prioritizing which technical debt should be tackled first. We map Mitre®'s 11 attack tactics to CWEs consequences. This mapping reveals which attack tactics can be used to exploit one of eight technical impacts caused by CWEs (detectable using static analysis), which currently includes 18 different CWEs. Traversing between attack tactics and CWE technical impacts helps us prioritize source code vulnerabilities that need attention to minimize technical debt.

1.1 Motivation and Research Objective

Although the usage of agreed upon CWEs as a basis for quantifying TD associated with security issues is a step in the right direction when providing meaningful quantification, it is not enough in a highly dynamic SecDevOps environment where organizations are under constant attack. A solution that ties adversarial behaviors to root causes in source code (i.e. vulnerabilities) is needed before said vulnerabilities are exploited (i.e. become weaknesses) causing technical debt interest that is not recoverable. This is an important distinction because our objective is to address the vulnerability associated with the attacker's behavior, not the results of static analysis tools usually executed out of context.

Further, agile and iterative SecDevOps approaches are seeing quick adoption in government organizations. According to the Congressional National Defense Authorization Act (NDAA)[7]: *"Not later than 30 days after the date of the enactment of this Act, the Secretary of Defense shall include the following systems for realignment under the pilot program to* ***use agile or iterative development methods*** *pursuant to section 873 of the National Defense Authorization Act for Fiscal Year 2018."* This represents a significant cultural shift in how software development and acquisitions is done in the federal government that affects a large number of programs. The SecDevOps approach embraces the congressional act and is being promoted by the Defense Acquisitions University (DAU)[8] with many trainings in place.

1.2 Contribution

Our position study provides the following contributions: *i)* a common link between the operational tactics employed by adversaries attempting to exploit a software system and the consequences of CWEs (i.e. technical impacts) and, *ii)* an approach to rank attack tactics used by adversaries based on how similar they are to an attack vector using the Jaccard Similarity Index ranking system [8]. The source code of the contribution is publicly available in a Github[9] repository.

2 BACKGROUND AND RELEVANT WORK

2.1 Technical Debt Quantification

A new definition for TD was presented by a group of academics and practitioners who participated in a Dagstuhl [9] in 2016. The definition was repurposed to be more focused and to help steer our community. Specifically:

"In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability."

A comprehensive synthesis of all approaches used to classify and quantify TD in the literature is beyond the scope of this paper; however, herein we describe the more notable approaches.

Tom et al. [10] identified many aspects of TD and classified them into five main components: code debt, design and architectural debt, environmental debt, knowledge distribution and documentation debt, and testing debt. The classification is broad but also abstract and allows for too many aspects to affect TD in a system. Tamburri et al. [11] also attempted to include socio-technical aspects of organizations as a form of TD.

Four prominent approaches to quantify TD are highlighted – all differ in their quantification. It is important to note that to the best of the author's knowledge, there are no approaches that quantify or prioritize TD as a result of behaviors observed by operations personnel such as cybersecurity first responders. SecDevOps environments would facilitate these observations thus allowing for quick turnaround and context relevant TD scoring.

Nugroho et al. [12] propose a formula to measure TD connected to the maintainability of software. No implementation of this approach is found in the literature. The formula focuses on the maintainability of software and gives a measurement of how much effort will be needed in order to repair the amount of TD in the software. A five-star rating scale is used to describe the quality of the maintainability in the system with one star signifying the lowest quality and five stars signifying the highest quality. TD is measured by multiplying a *rework fraction* and a *rebuild value*. The rework fraction is an estimated percentage of the number of lines in the code that contribute to the TD. The rebuild value is the estimated amount of time (*in months*) that needs to be spent fixing the TD.

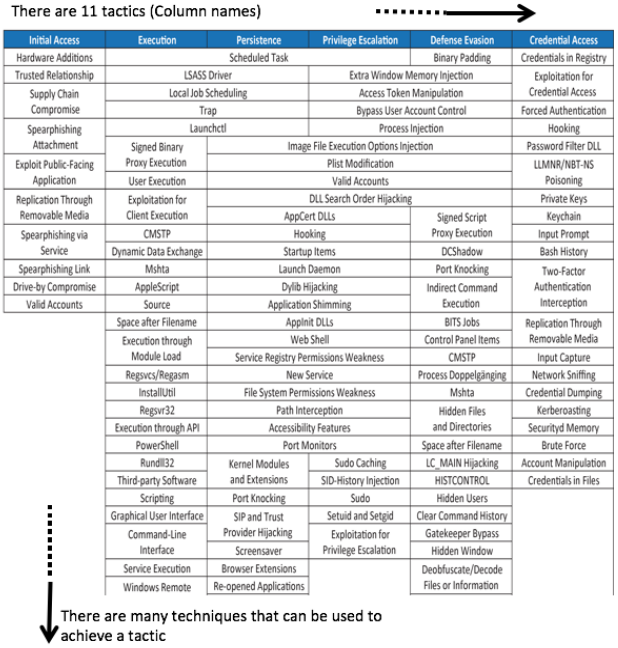
Letouzey and Ilkiewicz [13] use the SQALE method to estimate the amount of TD in a system based on an ISO quality model. The quality model uses a stack of eight quality features: testability, reliability, changeability, efficiency, security, maintainability, portability, and reusability. These features are organized in a pyramidal hierarchy where testability is at the

[6] https://www.sonarqube.org/
[7] NDAA Act, June 2018, section 891, sub section 873-874
[8] https://www.dau.mil/
[9] https://github.com/maryeprouty/attack-analysis

bottom and reusability at the top. The idea is that concerns at lower levels need to be addressed first before tackling issues at higher levels. This is necessary in order to effectively remediate issues. For example, a part of the code that does not meet a condition that is associated with testability should be addressed before one that is associated with maintainability.

The SonarQube[6] tool is quite popular amongst the community because they offer a free download of their framework which is composed of multiple widgets. One widget implements the calculation of TD and reports it in terms of days or dollars (i.e. cost) necessary to repay the debt.

Curtis et al. [14] introduced a way to measure TD that focuses on converting the amount of TD in code to a dollar amount. The formula used calculates TD principal by observing should-fix violations in the code, the estimated number of hours to fix the should-fix violations, and the estimated cost of labor to do so. Should-fix violations are classified to be either low-, medium-, or high-severity, and the formula assigns a higher weight to the higher severity violations and a lower weight to the lower severity violations in the formula. The principal is calculated by multiplying each level of severity by the number of violations that need to be fixed, the average number of hours it will take to fix them, and a dollar amount that represents the average cost per hour for work in IT organizations. A calculation is made for each level of severity to obtain three values, and the sum of the three values is used to calculate the TD principal.


There are 11 tactics (Column names)
There are many techniques that can be used to achieve a tactic

<table>
<tr><th>Initial Access</th><th>Execution</th><th>Persistence</th><th>Privilege Escalation</th><th>Defense Evasion</th><th>Credential Access</th></tr>
<tr><td>Hardware Additions</td><td colspan="3">Scheduled Task</td><td>Binary Padding</td><td>Credentials in Registry</td></tr>
<tr><td>Trusted Relationship</td><td colspan="2">LSASS Driver</td><td colspan="2">Extra Window Memory Injection</td><td rowspan="2">Exploitation for Credential Access</td></tr>
<tr><td rowspan="2">Supply Chain Compromise</td><td colspan="2">Local Job Scheduling</td><td colspan="2">Access Token Manipulation</td></tr>
<tr><td colspan="2">Trap</td><td colspan="2">Bypass User Account Control</td><td>Forced Authentication</td></tr>
<tr><td rowspan="2">Spearphishing Attachment</td><td colspan="2">Launchctl</td><td colspan="2">Process Injection</td><td>Hooking</td></tr>
<tr><td rowspan="2">Signed Binary Proxy Execution</td><td colspan="3">Image File Execution Options Injection</td><td>Password Filter DLL</td></tr>
<tr><td rowspan="2">Exploit Public-Facing Application</td><td colspan="3">Plist Modification</td><td rowspan="2">LLMNR/NBT-NS Poisoning</td></tr>
<tr><td>User Execution</td><td colspan="3">Valid Accounts</td></tr>
<tr><td rowspan="2">Replication Through Removable Media</td><td rowspan="2">Exploitation for Client Execution</td><td colspan="3">DLL Search Order Hijacking</td><td>Private Keys</td></tr>
<tr><td colspan="2">AppCert DLLs</td><td rowspan="2">Signed Script Proxy Execution</td><td>Keychain</td></tr>
<tr><td rowspan="2">Spearphishing via Service</td><td>CMSTP</td><td colspan="2">Hooking</td><td>Input Prompt</td></tr>
<tr><td>Dynamic Data Exchange</td><td colspan="2">Startup Items</td><td>DCShadow</td><td>Bash History</td></tr>
<tr><td>Spearphishing Link</td><td>Mshta</td><td colspan="2">Launch Daemon</td><td>Port Knocking</td><td rowspan="3">Two-Factor Authentication Interception</td></tr>
<tr><td>Drive-by Compromise</td><td>AppleScript</td><td colspan="2">Dylib Hijacking</td><td rowspan="2">Indirect Command Execution</td></tr>
<tr><td>Valid Accounts</td><td>Source</td><td colspan="2">Application Shimming</td></tr>
<tr><td rowspan="16"></td><td>Space after Filename</td><td colspan="2">AppInit DLLs</td><td>BITS Jobs</td><td rowspan="2">Replication Through Removable Media</td></tr>
<tr><td rowspan="2">Execution through Module Load</td><td colspan="2">Web Shell</td><td>Control Panel Items</td></tr>
<tr><td colspan="2">Service Registry Permissions Weakness</td><td>CMSTP</td><td>Input Capture</td></tr>
<tr><td>Regsvcs/Regasm</td><td colspan="2">New Service</td><td>Process Doppelgänging</td><td>Network Sniffing</td></tr>
<tr><td>InstallUtil</td><td colspan="2">File System Permissions Weakness</td><td>Mshta</td><td>Credential Dumping</td></tr>
<tr><td>Regsvr32</td><td colspan="2">Path Interception</td><td rowspan="2">Hidden Files and Directories</td><td>Kerberoasting</td></tr>
<tr><td>Execution through API</td><td colspan="2">Accessibility Features</td><td>Securityd Memory</td></tr>
<tr><td>PowerShell</td><td colspan="2">Port Monitors</td><td>Space after Filename</td><td>Brute Force</td></tr>
<tr><td>Rundll32</td><td rowspan="2">Kernel Modules and Extensions</td><td>Sudo Caching</td><td>LC_MAIN Hijacking</td><td>Account Manipulation</td></tr>
<tr><td>Third-party Software</td><td>SID-History Injection</td><td>HISTCONTROL</td><td>Credentials in Files</td></tr>
<tr><td>Scripting</td><td>Port Knocking</td><td>Sudo</td><td>Hidden Users</td><td rowspan="6"></td></tr>
<tr><td>Graphical User Interface</td><td rowspan="2">SIP and Trust Provider Hijacking</td><td>Setuid and Setgid</td><td>Clear Command History</td></tr>
<tr><td rowspan="2">Command-Line Interface</td><td rowspan="2">Exploitation for Privilege Escalation</td><td>Gatekeeper Bypass</td></tr>
<tr><td>Screensaver</td><td>Hidden Window</td></tr>
<tr><td>Service Execution</td><td>Browser Extensions</td><td rowspan="2"></td><td rowspan="2">Deobfuscate/Decode Files or Information</td></tr>
<tr><td>Windows Remote</td><td>Re-opened Applications</td></tr>
</table>

Figure 1: A partial view of Mitre's ATT@CK matrix

2.2 ATT@CK

The Adversarial Tactics, Techniques, and Common Knowledge framework is a knowledge base and a model for capturing adversarial behaviors and it reflects all the phases of the adversary's attack lifecycle. It is under the auspices of the Mitre® Corporation and aims to enumerate and categorize post-compromise adversary tactics, techniques and procedures against various operating systems. A tactic is at the core of the matrix and represents a high-level description of an attack behavior. Each tactic can be broken down into many techniques and procedures that an attacker may use to compromise a target system. The matrix has expanded to include other pre-compromise behaviors as well as mobile devices. It consists of three core components: *i)* 11 tactics (denoted by the columns in Fig. 1. The full matrix can be found in Mitre®'s website[5]), *ii)* 219 techniques that describe specific approaches used to achieve a tactical goal, and *iii)* documented adversarial usage techniques.

3 PILOT STUDY

We perform an attack-analysis simulation study that explores the landscape of potential techniques used by attackers that can be observed by operations staff in a SecDevOps environment.

Table 1: Attack tactic dependencies

Tactic	Dependency	Explanation
Persistence	Credential Access	This tactic is useful for attackers wishing to maintain their presence in the target network even in the face of loss of credentials
Execution	Initial Access Lateral Movement	An initial foothold into the target is necessary before adversary-controlled code and commands execution. In cases where the attacker cannot successfully compromise the system after initial access, the adversary will move across the network
Privilege Escalation	Lateral Movement	When attackers cannot gain privileges within an entry point, lateral movement is required
Exfiltration	Collection	An attacker will often need to be able to first gather the sensitive data in the system through "Collection" before it can be removed from the system
Collection	Discovery	Adversaries must gain an understanding of the system before gathering sensitive data
Command and Control	Discovery	Adversaries first employ discovery tactics to understand the system well enough to avoid detection during control activities
Defense Evasion		Adversaries employ this tactic to remain undetected

Specifically, we populate an attack vector $\vec{a}$ from observed behaviors and explore how it can relate to Mitre®'s CWEs. However, before computing a similarity score we performed an analysis of the various dependencies that exist between attack tactics. Due to the nature of Mitre®'s attack tactics, a simple bipartite graph relating tactics to CWE technical impacts is not sufficient. By manually examining the relationships between tactics, a clearer understanding of why and how tactics are used to exploit CWEs can be drawn. Many of the tactics depend on other tactics and have temporal precedence before they can be employed by an attacker. For instance, *Execution* depends on *Initial Access* so that the attacker can gain an initial foothold into the target network before executing their adversary-controlled code or commands. Table 1 shows a breakdown of tactics that are dependent on other tactics before they can be successfully employed by an attacker. Some dependencies are purely contextual, as in the case of *Privilege Escalation*'s dependency on *Lateral Movement* – if the attacker can gain privileges in the initial system, then there is no need to move across the network in order to employ this tactic. Note that *Defense Evasion* is not dependent on any other tactics nor do other tactics depend on it per se; however, this tactic is often used *in parallel* with other tactics.

Table 2: Technical Impacts associated with CWEs

CWE Technical Impact	Automatic Static Analysis	Manual Static Analysis
Execute unauthorized code	78, 79, 98, 120, 129, 131, 134, 190, 426, 798, 805	98, 120, 131, 190, 426, 494, 805
Gain privileges, assume identity	306, 352, 426, 601, 798	259, 306, 352, 426
Read data	78, 79, 89, 129, 131, 134, 352, 426, 798	89, 131, 209, 311, 327, 352, 426
Modify data	78, 89, 129, 131, 190, 352	89, 131, 190, 311, 327, 352
DoS: unreliable execution	78, 120, 129, 131, 190, 352, 400, 426, 805	120, 131, 190, 352, 426, 805
DoS: resource consumption	120, 190, 400, 770, 805	120, 190, 805
Bypass protection mechanism	79, 89, 190, 352, 400, 601, 798	89, 190, 352
Hide activities	78	327

After tactic dependencies were established, we investigated how tactics employed by adversaries map to technical impacts. Technical impacts are consequences of CWEs that negatively affect TD in a target. We can detect CWEs using static analysis techniques. Automated static analysis (i.e., FxCop [10] and FindBugs[11]) helped us detect 18 different CWEs. Manual analysis helped us identify 14 CWEs. Manual and Automated analysis overlapped on 9 common CWEs and helped us validate the automated findings. Table 2 relates the CWE numbers detectable from static analysis techniques to their technical impacts.

Finding a common link between the consequences of CWEs (i.e., technical impacts) and the tactics that are used to exploit a software system, allows developers to prioritize the TD associated with the vulnerabilities being exploited by the attacks. Thus, this mapping establishes which attack tactics can be used to impact TD caused by anyone of eight CWE technical impacts. Traversing from attack tactics to technical impacts provides a way to connect the detected tactics employed by attackers to the CWEs associated with source code vulnerabilities. This allows developers immediate access to TD prioritization based on operations experiences. Fig. 2 displays the tactic dependencies graph, and does not illustrate parallel usage of tactics. Future work could investigate which tactics are most often used in conjunction with one another. Fig. 3 shows the mapping of tactics to technical impacts.

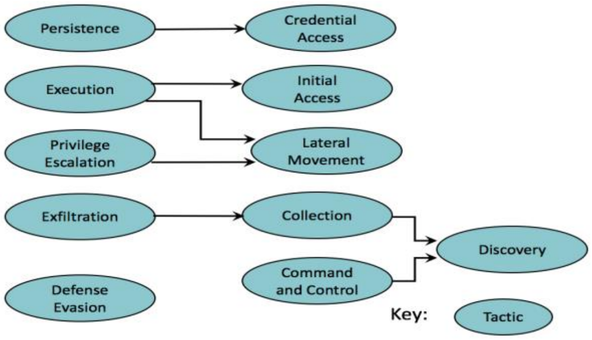


Figure 2: Attack Tactics Dependencies

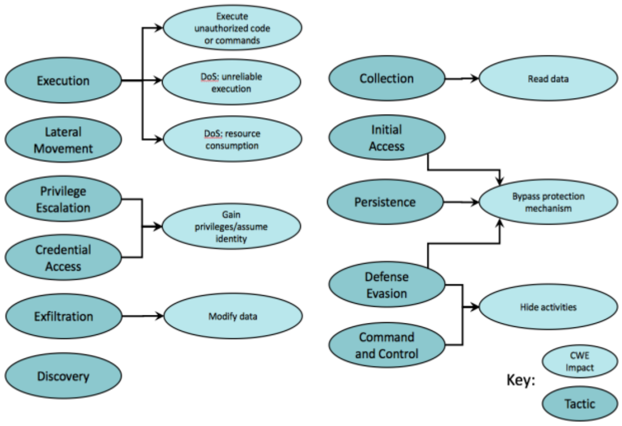


Figure 3: Mapping of attacker tactics to CWE impacts

Once the tactic dependencies and the mappings from technical impacts to CWEs were agreed upon, the following steps were followed:

1. A file of n randomly simulated attack vectors, each containing up to m techniques, defined as $\vec{a}_{ij}$ where $i = 1..n$ and $j = 1..m$ is compared to each of the tactics from the ATT@CK matrix,
2. For each attack, the ATT@CK tactics are ranked based on how similar they are to the simulated attack vector $\vec{a}_{ij}$ using the Jaccard Similarity (JS) Index ranking system,
3. The graphs are traversed from the top ranked JS tactic to technical impact; which suggests the CWEs that are most likely to be at risk of attack based on the attack vector $\vec{a}_{ij}$. The tactics dependency graph also points to secondary potential CWEs.
4. CWEs describe the source code that requires attention.

[10] https://www.microsoft.com/en-us/previous-versions/dotnet/netframework-3.0/
[11] http://findbugs.sourceforge.net/

To use the JS Index, we convert the ATT@CK matrix to csv format where a 1 represents the partake of a technique in that tactic and a 0 does not. Attack vectors $\vec{a}_{ij}$, are generated randomly where $\vec{a}_{ij} = 1$ indicates that a technique is used/detected, $\vec{a}_{ij} = 0$ indicates that a technique is not used/detected, and $\vec{a}_{ij} = ?$ indicates that a technique is not detectable. We use the '?' symbol in order to simulate cases where a system cannot detect certain techniques; which is a common occurrence. By using a '?' instead of a 0 in these situations, the algorithm for JS is not skewed by the system's inability to detect the techniques, and instead can compute the closest tactic to the attack vector using available information. The comparison of a simulated attack vector $\vec{a}_{ij}$ against a tactic vector $\vec{t}_{ij}$ of the ATT@CK matrix uses the JS index, which measures the binary overlap between the attributes of two vectors $\vec{a}_{ij}$ and $\vec{t}_{ij}$. Equation $JS = M_{11}/(M_{01} + M_{10} + M_{11})$, where M_{11} is the total number of attributes where $\vec{t}_{ij}$, and $\vec{a}_{ij}$ is 1, M_{01} is the total number of attributes where $\vec{t}_{ij}$ is 0 and $\vec{a}_{ij}$ is 1, M_{10} is the total number of attributes where $\vec{t}_{ij}$ is 1 and $\vec{a}_{ij}$ is 0, and M_{00} is the total number of attributes where both vectors equal 0, yields a similarity score for any two vectors. Note that for any two vectors $\vec{a}_{ij}$ and $\vec{t}_{ij}$, $M_{01} + M_{10} + M_{11} + M_{00} = m$.

The simulation algorithm traverses the dependency and association graphs from the highest ranked tactic to determine which technical impacts are most at risk of being exploited by this tactic. Each technical impact has several CWEs associated with it; thus, an attack vector can be analyzed to determine which CWEs are most vulnerable to a given attack. This ranking allows developers to address the TD associated with code vulnerabilities as a result of real attacks observed by operations staff. Two csv files are compared, where one is a file of Mitre®'s attack tactics and the other is a file of simulated attack vectors. It ranks the 11 tactics for each simulated attack vector using JS, and then traverses the graphs to output the most vulnerable CWEs to each attack. A Swing application provides a visualization of these graphs for the user to view and interact with.

4 POSITION ON TECHNICAL DEBT

In the context of SecDevOps environments we are afforded a unique opportunity to address cybersecurity threats to computational environments quickly, and the decisions that developers can make to address the technical debt associated with said systems are vastly improved because of context – Ops is in constant communications with Devs. Today, we run static analysis tools to detect source code disharmonies and to compute the TD principal associated with source code, but we often run these tools independently of any other lifecycle phases or Ops, and many times developers are not aware of the TD in the source code until they review relevant QA reports. This disconnection affects the decisions that developers make in terms of prioritizing which debts to fix first. Executing static analysis tools out of context does not help operational staff because although first responders may be able to mitigate an attack, the TD associated with the relevant source code vulnerabilities may still persist.

By using an approach that can quickly map an attack to a relevant CWE, developers can prioritize much more effectively and fix the source code responsible for the vulnerability. It is also our position that the longer a technical impact associated with an attack goes unattended, the larger the TD interest incurred.

Thus, our approach allows for:

i. addressing $\text{Principal}_{\text{TD-Security}}$ in context, and

ii. reducing the $\text{Interest}_{\text{TD-Security}}$ because relevant issues are tackled quickly

6 CONCLUSION AND FUTURE WORK

In line with our prior conclusions [2], it is our position that security is a special case because the TD associated with cybersecurity cannot just be measured in terms of maintainability, but also in terms of damage to a business. Addressing the TD needs to occur quickly in context with Ops. SecDevOps allows developers an opportunity to address TD dynamically. Significant work remains in terms of industrial and open source studies.

ACKNOWLEDGMENTS

This research is funded by the Construction Engineering Research Laboratories (CERL), Air Force, and the Department of Defense through an intermediary partnership with TechLink. Funding is also provided by NSF grant 1658971.

REFERENCES

[1] A. A. U. Rahman, L. Williams, "Software Security in DevOps: Synthesizing Practitioners' Perceptions and Practices", Proc. of the 2016 Intl. Workshop on Continuous Software Evolution and Delivery (CSED), May 2016.

[2] C. Izurieta, D. Rice, K. Kimball, T. Valentien, "A Position Study to Investigate Technical Debt Associated with Security Weaknesses," Proc. of the 2018 Int. Conference on Technical Debt (TechDebt), May 2018.

[3] Software Product Evaluation—Quality Characteristics and Guidelines for Their Use, ISO/IEC Standard ISO-9126, 1991

[4] "ISO/IEC 25010:2011 Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – Systm and Software Quality Models," Mar. 2011.

[5] S. Wagner, K. Lochmann, L. Heinemann, M. Klas, A. Trendowicz, R. Plosch, A. Seidi, A. Goeb, and J. Streit, "The Quamoco product quality modelling and assessment approach." IEEE, Jun. 2012, pp. 1133–1142.

[6] M. G. Siavvas, K.C. Chatzidimitriou, A. L. Symeonidis, "QATCH – An adaptive framework for software product quality assessment," Journal of Expert Systems with Applications, Vol. 86, pp. 350-366, Elsevier, Nov. 2017.

[7] I. Griffith, C. Izurieta, and C. Huvaere, "An Industry Perspective to Comparing the SQALE and Quamoco Software Quality Models," IEEE ACM 11th International Symposium on Empirical Software Engineering and Measurement (ESEM). Toronto, Canada, Nov. 9-10 2017.

[8] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura", Bulletin de la Société Vaudoise des Sciences Naturelles, 1901, 37: 547–579.

[9] Managing Technical Debt in Software Engineering, Dagstuhl Reports, Vol. 6, Issue 4, April 17-22, 2016. [Online] Available: dagstuhl.de/16162

[10] E. Tom, A. Aurumn, and R. Vidgen. "An Exploration of Technical Debt," Journal of Systems and Software, Vol. 86, Issue 6, pp. 1498-1516, June 2013.

[11] D. Tamburri, Philippe Kruchten, P. Lago, and H. van Vliet. "What is Social debt in Software Engineering," CHASE 2013, San Francisco USA.

[12] Nugroho, A.; Visser, J.; Kuipers, T., "An empirical model of technical debt and interest," In Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11). ACM, New York, NY, USA, 1-8. doi:10.1145/1985362.1985364

[13] J.L. Letouzey and M. Ilkiewicz, "Managing Technical Debt with the SQALE Method," IEEE Software Vol. 29, Issue 6, Nov-Dec 2012

[14] B. Curtis, J. Sappidi, and A. Szynkarski. "Estimating the principal of an application's Technical Debt," IEEE Software Software Vol. 29, Issue 6, Nov-Dec 2012, IEEE doi:10.1109/MS.2012.156