A Grounded Theory Based Approach to Characterize Software Attack Surfaces

Sara Moshtari
Rochester Institute of Technology
Rochester, NY, USA
sm2481@rit.edu

Ahmet Okutan
Rochester Institute of Technology
Rochester, NY, USA
Axoec@rit.edu

Mehdi Mirakhorli
Rochester Institute of Technology
Rochester, NY, USA
mxmvse@rit.edu

ABSTRACT
The notion of Attack Surface refers to the critical points on the boundary of a software system which are accessible from outside or contain valuable content for attackers. The ability to identify attack surface components of software system has a significant role in effectiveness of vulnerability analysis approaches. Most prior works focus on vulnerability techniques that use an approximation of attack surfaces and there has not been many attempt to create a comprehensive list of attack surface components. Although limited number of studies have focused on attack surface analysis, they defined attack surface components based on project specific hypotheses to evaluate security risk of specific types of software applications. In this study, we leverage a qualitative analysis approach to empirically identify an extensive list of attack surface components. To this end, we conduct a Grounded Theory (GT) analysis on 1444 previously published vulnerability reports and weaknesses with a team of three software developers and security experts. We extract vulnerability information from two publicly available repositories: 1) Common Vulnerabilities and Exposures (CVE), and 2) Common Weakness Enumeration (CWE). We ask three key questions: where the attacks come from, what they target, and how they emerge, and to help answer these questions we define three core categories for attack surface components: Entry points, Targets, and Mechanisms. We extract attack surface concepts related to each category from collected vulnerability information using the GT analysis and provide a comprehensive categorization that represents attack surface components of software systems from various perspectives. The comparison of the proposed attack surface model with the literature shows in the best case previous works cover only 50% of the attack surface components at network level and only 6.7% of the components at code level.

KEYWORDS
Software Security, Attack Surface, Grounded Theory, Qualitative Analysis

1 INTRODUCTION
With thousands of new vulnerabilities being discovered every year, software security is more increasingly becoming a day-to-day concern for organizations across the world. Software security practitioners use a wide range of security analysis techniques to improve the confidentiality, integrity, and availability of software systems. These techniques often directly or indirectly rely on understanding application’s attack surfaces—a set of points on the boundary of a software system, where an attacker can try to enter, cause an effect on, or extract data from, that system[11, 17].

Attack surface analysis—the process of identifying applications’ attack surface components (a.k.a points) play a key role in numerous methods for security risk analysis [10–12, 17, 19], vulnerability detection [2, 13–15, 25, 26, 30, 31, 33], and software testing [1]. Prior software vulnerability detection and testing approaches consider Sink, Source, Entry point, and API/Function calls as parts of the attack surface, however these studies primarily focus on the the analysis itself, rather than identifying attack surface components. There are a few studies that elaborate on the notion of the attack surface [10–12, 17, 19, 25], consider entry points, exit points, channels etc. as attack surface components, and test and validate them as existing theories [11, 12, 18, 19, 25]. They focus on limited-scope and example based demonstration of attack surfaces of operating systems [11, 17, 19], and web applications [18] using the attack surface metaphor. These studies show that applications with smaller attack surface are less vulnerable. While there has been a significant interest by practitioners and in the literature to study attack surface components of a given system, unfortunately, we lack a generic comprehensive guidance to support security research engineers in identifying attack surfaces of a given system. To the best of our knowledge there is no prior research that takes a comprehensive approach to characterize and identify attack surface components in software systems.

In this paper we take a Grounded Theory (GT)-based approach [8, 9, 28] to characterize software attack surfaces and develop a comprehensive attack surface model to be reused by researchers and practitioners. Grounded theory is a qualitative approach that extracts theories from unstructured data and leads to discoveries directly supported by empirical evidence [8, 9, 28]. We extract and analyze 810 vulnerability reports from Common Vulnerabilities and Exposures (CVE) data published by U.S. National Vulnerability Database (NVD)[20]. In addition we analyze 634 entries in Common Weakness Enumeration (CWE) data, an extensive catalog of different types of software and hardware weaknesses describing root causes of vulnerabilities [21]. We leverage the Grounded Theory to extract high-level concepts which are related to software systems’ attack surface from vulnerability reports and weaknesses, and use Straussian GT [6, 7] as a systematic inductive method for conducting qualitative research of identifying attack surface components. Our GT analysis starts by asking three key research questions:

- Research Question 1: Where are the critical entry points in a software system that are used by attackers to get in?
- Research Question 2: What assets or components in a software system are targeted by attackers?
- Research Question 3: How attack surfaces emerge, and what types of mechanisms are utilized to reach to targets?
To answer these research questions we consider three core theories related to each research question, which are Entry Points, Targets, and Mechanisms, respectively. We extract the concepts related to each theory from gathered vulnerability data and define a generic attack surface model based on the emerged concepts. During the GT process we find that concepts related to each theory can be categorized in four major groups: software source code (Code), its executable (Program), the System, and the Network environment. Then, we identify attack surface components under each category and compare proposed attack surface categorization model with the literature. The comparison results indicate that our study has discovered 280 components, almost all attack surface components defined in the literature are covered by the proposed attack surface model, while prior works cover only a small portion of the concepts identified by our analysis. The result of quantitative comparison show that in best cases 50%, 30%, and 18.7% of the low level concepts which are related to Network, System and Program level categories are covered by the literature. At Code level, in best case, 6.7% of concepts are covered by the literature.

The remainder of this paper is organized as follows: Section 2 provides an overview of the methodology used in this empirical study, Section 3 discusses the findings of our research, Section 4 compares our results with those in the literature, Section 5 discusses the results, and Section 6 provides concluding remarks.

2 METHODOLOGY

Attack surface refers to the amount of code, functionality, and interfaces of a system exposed to attackers[12]. In this study, we rely on publicly available vulnerability repositories to identify common attack surface components in software systems. Vulnerability databases describe vulnerabilities using natural language and do not include technical data. In order to identify attack surface components, we use an approach based on the Grounded Theory (GT) [8, 9, 28]. Since our research is led by research question we use the Straussian GT rather than Classic Glaserian GT [6, 7]. The Straussian GT is preferred, because we use existing concepts in the literature during our analysis [27, 29]. The Straussian grounded theory encompasses the following activities: (1) defining research questions, (2) theoretical sampling, (3) open coding, (4) constant comparisons, (5) memoing, and (6) axial coding and (7) selective coding. Figure 1 shows how we applied the Straussian grounded theory to our research that was conducted over the period of one year.

2.1 Research Questions

In Straussian grounded theory approach [3], researchers may define research questions upfront. While following the approach, we consider broad and open-ended research questions for detecting attack surface components. Getting inspired from the apparent attack surfaces of a house, research questions are defined based on the past experiences of the authors and concepts from the literature. For example, in a house, front and back doors, windows, garage door, climbable trees or tables can be entry points and the attacker would consider precious items in a house such as safety box as target. There might be some mechanisms in building a house such as emergency stairs that could make the house more vulnerable. Using the similarities of a software system and a house from the perspective of an cyberattacker, we identify the concepts for software applications to help define attack surface components. We focus on three research questions listed in Section 1 during the GT process and try to do coding in a way that can find theories from data to answer these questions.

2.2 Data Collection

Given the topic of interest of this study, we need access to software vulnerability reports, the description of these vulnerabilities, in-depth analysis of how they occurred, as well as vulnerable code snippets and their patches. Thus, we targeted open data resources that contain different types of vulnerabilities and vulnerable code snippets.

2.2.1 Theoretical Sampling. It is the data collection process that is based on the concepts derived from data [3, 29]. In theoretical data sampling, unlike conventional approaches, all data are not collected at the beginning. Data collection and analysis is a circular process. Concepts that are identified in each cycle lead to more data collection until the saturation occurs.

2.2.2 Data Sources. We obtained vulnerability information from two publicly available vulnerability repositories:

- **Common Weakness Enumeration (CWE):** CWE enumerates a list of common security weaknesses and categorizes them based on different views to help practitioners in securing their applications. It provides a concise description of the weakness, common consequences, likelihood of exploitation, demonstrative examples, and reference to other resources. We reviewed all these information pieces to extract attack surface components.

- **Common Vulnerabilities and Exposure (CVE):** The Mitre corporation CVE is an open platform to list publicly disclosed vulnerabilities. We used the CVE list and additional information provided in the National Vulnerability Database (NVD) to extract vulnerability meta-data.

We used issue tracking systems to obtain further discussions about CVEs, source code repositories to identify fixes for vulnerabilities, and further resources to extract other related information if existed. The summary of the data sources used:

1. **Retrieve vulnerabilities from MITRE and NVD:** We obtain vulnerability reports from MITRE CVE and NVD by consuming their public data feeds. Vulnerabilities disclosed in CVE are assigned a unique Identifier (CVE ID), a concise description, a list of affected software releases, and a list of references that can be used to obtain further details about the CVE, such as Issue Tracking Systems.

2. **Obtain vulnerability details from issue tracking systems:** Although CVE reports provide information about different attributes of a vulnerability, they do not contain enough information to identify attack surface components at code level. Thus, we reviewed associated issue tracking systems for vulnerabilities that are related to open source projects. We leveraged the list of “references” to identify URLs to the corresponding bug entry of the issue tracking system and we read the developers’ discussion about the problem, original code fragments, and their proposed solution(s).
3 Gather patches from code repositories: To retrieve patches that fixed vulnerabilities, we gathered the commits whose message explicitly mentioned the related bug id in the issue tracking systems or directly referred to the associated CVE. These patches often contained more information about the vulnerability, and the files that were affected, i.e., modified, added or removed during the fix. Identifying patches helped us to identify entry points, targets and mechanism at the code level.

4 Collect vulnerability details from other references: In addition to information that are provided in CVE website, we analyzed all the URLs that are provided as references for each vulnerability. These references include links to vulnerability reports, advisories or exploit information. These references provide more information for attack surface analysis of the vulnerability.

5 Get vulnerability details from related CWEs We identified related CWEs for each vulnerability. The CWEs helped us to understand the security issue of the CVE and extract attack surface components.

2.2.3 Data Collection Process. Number of vulnerabilities reported by NVD has increased from 6500 in 2016 to above 18000 in 2020 [16]. We focus on the vulnerabilities that have been reported during last five years (from 2016 to 2020 inclusive) to cover different types of weaknesses. For a more comprehensive attack surface analysis, we also collect data from CWE [21] and Introduced During Design/Implementation view [4, 5].

First stage: At the beginning, we selected 100 random vulnerabilities from each year (a total of 500) and collected their attributes. For some of these vulnerabilities only a CVE description was available (without any patch or advisory info), therefore, we were able to collect limited amount of data for such vulnerabilities. During the first stage of data collection, we performed coding process and extracted initial concepts.

Second stage: During this stage, we randomly selected 200 CVEs from 2016-2020 but omitted the areas that were saturated during the first stage of the analysis. As we reviewed the CVEs, we didn’t do coding for the CVEs that were related to vulnerabilities such as SQL injection, Buffer Overflow, Cross-site scripting and Command injection (36 CVEs), because we could not find any new information about attack surface components of these types of vulnerabilities in late steps of previous stage. We noticed that CVEs collected during the first stage, covered limited number of CWE branches (70 CWE IDs), therefore we also collected data from CWE and Introduced During Design and Introduced During Implementation views (which totally contain 634 weaknesses after removing their common CWE IDs) [4, 5] to be more comprehensive. During our analysis in the second stage, we extracted all components that can be part of an Entry Point, Target or Mechanism. Figure 2 shows the information model of the vulnerability data collected.

Third stage: In this stage, to identify new CVEs for emerged concepts that seemed incomplete and needed further analysis, we performed keyword-based selection of CVEs. We selected 110 CVEs related to these concepts. For example, for program architecture category we searched CVEs based on “Architecture”, “Model”, “Event Driven”, “Master Slave”, “Client Server”, and etc.

2.3 Open Coding

After data collection, we performed open coding on the vulnerability data[3, 9]. The open coding process analyzes collected data for each vulnerability and annotates them with codes (concepts). We review the information gathered for each vulnerability (description, discussions, exploitation mechanism, and patches etc.), annotate the information that are related to an Entry Point, Target or Mechanism, and assign a code to the annotated key points. A code is a phrase that summarizes the key point in a descriptive way. Extracted codes are assigned to the three general groups of attack surface components based on their relevance. Figure 3 shows how data was collected for CVE-2020-7284 as an example. In addition to the information available in MITRE website, we collected data from its security advisory (for exploit information), related GitHub repository (for source of code), and CWE. The key points that are highlighted in red were extracted from both code and descriptions and assigned to the Entry Point, Target and Mechanism:

- Entry Point: service request, ubus command, binary blob/json (input data), procedure call, message handler,
2.4 Constant Comparisons

We annotated vulnerability reports either by using the existing codes or creating new ones (if existing codes were not suitable for a newly analyzed vulnerability report). During the analysis of vulnerability reports, we also compare the existing concepts/categories against vulnerability reports to evolve categories and data interpretations. The overall goal of the open coding and constant comparison analysis is to identify the core concepts and categories related to the attack surface.

2.5 Memoing

Memos are notes, diagrams, or sketches that aid researchers to describe their preliminary ideas about properties and conceptual relationship between categories. After memoing, the researcher has stacks of memos in hand and puts them in an organized order by doing memo sorting [3]. Memoing is performed throughout the entire process of data coding and categorization. We used mind map diagrams to show the relationship between codes/concepts to identify core categories.

2.6 Axial Coding

During axial coding, we define new codes as a result of identifying new relationship and links [3, 29], and categorize the codes based on their relationship into higher level concepts. We perform axial coding for each research question separately. For instance, we can categorize codes gathered from CVE-2020-7248 and some others CVEs (e.g. CVE-2016-9424) for the Targets into two higher level concepts: 1) Application source code (memory manipulation statement), and 2) Application resource (memory). In this step, we also returned to the CVE instances in order to further refine and capturing all attack surface components.

2.6.1 Data Analysis Instruments. For the GT analysis, we developed a custom-built Web-based tool to support our coding activities. This tool presents the information retrieved for each vulnerability report based on the research questions.

2.7 Selective Coding

In the last step of our analysis, we finalize the categorizations by sorting the extracted concepts and associating them with the central branches, i.e. the Program (P), Code (C), System (S) and Network (N) [29]. During this process, we integrate previously identified

---

**Figure 3: Open coding of data collected for CVE-2020-7248**

| CVE-2020-7248 |
|----------------|
| **Description:** libubox in OpenWrt before 18.06.7 and 19.x before 19.07.1 has a tagged binary data JSON serialization vulnerability that may cause a stack based buffer overflow. |
| **References:** [https://github.com/openwrt/openwrt/commits/master](https://github.com/openwrt/openwrt/commits/master) |
| **Related CWE:** CWE-787: Out-of-bounds Write |

| Security Advisory 2020-01-31-2 |
|----------------|
| **Description:** Possibly exploitable vulnerability exists in the libubox library of OpenWrt, specifically in the parts related to JSON conversion of tagged binary data, so called blobs. An attacker could possibly exploit this behavior by providing specially crafted binary blob or JSON which would then be translated into blob internally. This malicious blobmsg input would contain blob attribute fielding large enough numeric value of type double then which processed by blobmsg_format_json would overflow the buffer array designated for JSON output allocated on the stack. The libubox library is a core component in the OpenWrt project and utilized in other parts of the project. Those interdependencies are visible by looking up of the above mentioned vulnerable blobmsg_format_json function in the project’s LXO[1], which reveals references in netifd, proof, usb, sysctl, udhcpc... libubox in OpenWrt before 18.06.7 and 19.x before 19.07.1 has a tagged binary data JSON serialization vulnerability that may cause a stack based buffer overflow. |
| **Exploit Info:** In order to exploit this vulnerability, a malicious attacker would need to provide specially crafted binary blobs or JSON input to blobmsg_format_json, then creating stack based overflow condition during serialization of the double value into the JSON buffer. It was verified, that its possible to crash by following shell command: `ubus call luci.getFeatures | "bash": /bin/sh -c "echo "status_code": similar to evolution of application resource (memory). In this step, we also returned to the CVE instances in order to further refine and capturing all attack surface components. |

---

| OpenWrt Github |
|----------------|
| **packages/utils/aes/src/aes.c:** |
| 39 include `<libcrypto.h>` |
| 40 include `<libubox/libubox.h>` |
| 41 include `<libcrypto/aes.c>` |
| 679 defines the status_code: 0 (ok), 1 (error), 2 (status_code). |
| 688 `DPRINTF[1]("attr": "blobmsg_format_json_indent (id):");` |
| 584 blobmsg_passheader, policy..., _H_MAX, th, blob data(c) - > meta, blob_len) > meta);` |

| OpenWrt.org Cross Reference |
|-----------------------------|
| **libubox/blobmsg_json.h:** |
| 40 static inline char *blobmsg_format_json_indent(struct blob_attr *attr, bool list, NULL (NULL indent);) |
| 41 | return blobmsg_format_json_with_cb(attr, list, NULL (NULL indent);) |
| 42 | return blobmsg_format_json_with_cb(attr, list, NULL (NULL indent);) |
| 130 | blobmsg_format_json_with_cb(attr, list, NULL (NULL indent);) |
| 322 char *blobmsg_format_json_with_cb(struct blob_attr *attr, bool list, |
| 323 blobmsg_format_json_with_cb(struct blob_attr *attr, bool list, |
| 354 ... state blob blobmsgprotobuf struct s, const char *c, int len) |
| 131 | status_code); |
| 132 char *new_buf; |
| 133 | if (len <= 0) |
| 134 | return true; |
| 135 | return true; |
| 136 | return true; |
| 137 | return true; |
| 138 | return true; |

---

| C2E-787 |
|----------------|
| **Description:** The software writes data past the end, or before the beginning, of the intended buffer. Example: In the following example, it is possible to request that memory move a much larger segment of memory than assumed (bad code). |

---

**Table:**

| Codes → Entry point: | Target: Memory copy state-ment, "memcpy() function" | Mechanism: Serialization/Deserialization |

---

**Table:**

| Codes → Entry point: | Service request, "unbus command", "binary blob/json (input data)", "procedure call", "message handler", "Target: Stack memory", "buffer array" | Mechanism: Serialization/Deserialization |

---

**Table:**

| Codes → Entry point: | Target: Binary blob (input data), "memory copy statement", "Mechanism: Serialization/Deserialization" |
3 RESULTS OF GROUNDED THEORY ANALYSIS

Based on the concepts emerged at the end of our analysis, we find that each core category Entry Point, Target, and Mechanism can be divided into four major groups, i.e., Code (C), Program (P), System (S), and Network (N). From Where the attackers are entering into a system (entry points), what they are targeting for (targets) and how they are executing their attacks (mechanisms) are all related to the source Code of a software, its executable version named Program, the System that application is installed on, and the Network that the system is interacting with. The results of this study learned from reviewing CVEs and CWEs are presented based on three research questions associated with Entry Points, Targets and Mechanisms:

3.1 Entry Points

Figure 4 shows the key categories we define for entry points based on the emerged concepts at the end of our analysis to answer to Research Question #1. We define entry points based on four core categories:

3.1.1 Code. This category represents parts of source code that an attacker can leverage to enter a system. As shown in Figure 4 they are categorized into three sub categories:

1. User Interface (UI) defines components in the UI that can be used by attackers to enter a system. For example, an attacker can interact with an application through components in the graphical user interface (Input box, File Upload[10], RSS Feed[10], etc.) or Console.
2. Methods/Directives defines methods or directives that receives input. They can be part of the code that directly receives input (Direct Entry) such as Input Methods[19, 25] that receives inputs directly from User, Device, or File or Handlers that handle different requests such as OS Signal (Interrupts) or web requests (REST API, Java Servlet). Indirect Entry covers parts of the source code that indirectly receives input by loading Code, reading Indirect Inputs (such as Environment Variable, System Attributes etc.) or User Created Resources.
3. Configuration Files category contains accessible configuration files of an application that can act as an indirect entry point for software application.

3.1.2 Program. This category considers an application as an executable and defines attack surface components related to that:

1. Components refer to special software components that open the doors for attackers, such as application components which are designed during the design phase of software development. For example, Plugin, Installer Components, Chatting component, and Authentication/login components are software components that can be considered as entry points at the design level.
2. Maintenance category covers any action that is performed during Deployment or Maintenance of programs that can open the doors for attackers. Install, Configuration, and Update operations are defined in this category.
3. Direct/Indirect Input category covers data that is sent to the program. Application can receive Direct Input[19] from User [25], Device, Operating System (OS), or as Messaging Object from other components/applications (Intent in Android). It can also receive Indirect Input by reading/loading Environment Variables, DLL Files, OMX buffer, System Properties, Virtual Machine Properties, and Cookies[10].

3.1.3 System. As a platform for running software applications, can provide entry points:

1. System input contains both Direct Input and Indirect Input. Direct Input represents the requests that are sent to the system and is categorized into Connection Requests (SSH request), OS Commands and Service Requests. Indirect Inputs represent the types of data imported by the system such as Load Driver. 2. Access Control contains actions that may open the door for attackers. It contains Local Access to the system or Improper Access Control.

3.1.4 Network. category contains the Packet, Socket and Access Control sub categories:

1. Packet represents the input data at the network level.
2. Socket[12, 17, 19], Port, and Protocols[10, 12, 19] could provide entry points at the network level.
3. Access Control contains actions the open the door for attackers such as Local Access to the network.

3.2 Targets

Figure 5 represents the categorization model created for Targets to answer to Research Question #2.

3.2.1 Code. This category defines source code related components that can be target of attacks. Attacker might try to access parts of source code to do malicious action. As shown in Figure 5 these components are categorized into two categories:

1. User Interface (UI) category refers to components in the user interface that can be target of an attack. The analysis identified target components in this category such as Validators and HTML/Webscript that are related to Web-based applications.
2. Method/Code Fragment represents methods or other related parts of the code that can be target of attacks. As shown in Figure 5, parts of source code that handle requests (Handlers), execute Commands (Database or OS [25]), do Memory Manipulation, Serialization/Deserialization, Reflection, Dangerous Operations such as Type Casting, Integer Operations, Coding/Encoding etc. can be attractive targets for attackers.

3.2.2 Program. The concepts under Program are categorized into two general categories:
1. Resource contains resources allocated or used by the application such as Memory, Stack, Heap, Cache, Shared Memory [12] and other memory types that are allocated, used, or read by the application.
2. Data covers important application data that are identified as target during GT analysis. This category considers application data from two perspectives: 1) Data Resource which represents the location where data is stored (Database, File etc.), and 2) Sensitive Information that represents various kinds of important data that an attacker may look for.

We found important files that can be target of attacks such as Lock File, Log File, Certificate, and Keystore File. Credentials such as User, Used Service (Notarization Service), and Database credentials.
3.2.3 System. This category defines components in the OS or firmware that can be target of attacks. They can be directly accessed through attacks against the system or indirectly through attacks against software programs that use these components. These components are categorized into two major categories: (1) Operating System (OS) contains OS and server related target components. They are categorized into different abstraction levels. System Availability covers actions that affect the availability of systems. For instance, a malicious System Reboot could interrupt a system and affect its availability. System Data categorizes different types of data in a system that can be target of attacks. It categorizes System Data based on the resource it is stored (Data Resource and the type of the data (Sensitive Information). Data Resources can be a database on the OS (like Windows Registry[12, 19]) or an important File/Directory on the file system. Critical Directory may contain sensitive information (like etc/passwd in Linux), Symlinks/Shortcuts (like Unix Hard Link or Symbolic Link [17], Windows Shortcut), File System Specific Files (like Data/Resource Fork of a File in HFS+ file system, Alternate Data Streams (ADS) in NTFS file system etc) and System/Server Critical Files (WSDL File in Web Server, Zone File in DNS Server, Node Catalogue in Distributed System etc). User Account information, Process Information, and Connection Pool are Sensitive Information in a system. Services/Server defines types of servers or services on a system that are usually target of attacks. For instance, SSL and NAS Servers are identified as targets in various vulnerability reports [22, 24].

2) Firmware. category covers parts of firmware that contain Device Information or control Device.

3.2.4 Network. 1) Packets and information on 2) Network Devices OS such as Process (Routing Engine on Routers), Device Setting, and Device Data and also 3) Socket Buffer could be the target of attacks at the network level.

3.3 Mechanisms
This category answers the Research Question #3 by discussing mechanisms that are used at the source code, program, system, and network level that could lead to the emergence of vulnerable attack surfaces. Figure 6 represents the categorization model for Mechanism. We briefly summarize the mechanisms:
3.3.1 Code. Mechanisms used at the code level are categorized in three major categories:

1. **User Interface (UI)** defines mechanisms used in the UI to open the doors for attackers. The concepts under this category are related to Web-based applications. Using Unsafe Techniques such as CSS Filters, RSS Feeds[10], Active Content[10], and Web Widget.

2. **Methods/Code Fragments** category discusses vulnerable mechanisms used during coding such as Using Third-party Library, Serialization/Deserialization, Polymorphic Deserialization, Improper Security Check, and Authentication. Improper Security Check focuses on security mechanism that are missed or implemented incorrectly such as Improper Input Validation[10], Weak Encryption, Load Multiple Certificates (like system and SSL certificate), and Number of Requests. Authentication category contains Authentication Methods (Password, Token-based, Certificate-based etc.), and Authentication Implementation mechanisms such as Location (No Server Side Authentication) and Insecure Implementation (use == operator instead of === for Hash Comparison or unsafe info such as IP address for Authentication). 3. **Development Framework** category discusses vulnerable mechanisms which are related to the software development frameworks, like Improper Configuration (Creating Debug Binary or Improper Encryption), and Installing Package Installers.

3.3.2 Program. This category categorizes the mechanism that are related to the design and deployment phase of a software system: 1. **Design** category discusses mechanisms related to the design such as Program Architecture (Client/Server etc.), Interaction (connecting to Other Servers or Other Applications), and insecurely designed Workflow. 2. **Deployment** category covers mechanisms during program deployment such as Improper Configuration and Unsafe actions that users can do on a system such as Install Add-on and Enable Dangerous Features[12].

3.3.3 **System Level.** category covers the following concepts:

1. **Dangerous Services/Server** category refers to activating special Servers (e.x. SSL, NAS, and Mail) or installing/activating Unsafe Services/Server (like Unsafe FTP server) that can lead to emergence of attacks. 2. **Dangerous Program** refers to installing special programs that can open the door for attackers (e.x. Android application that allows disabling/enabling WIFI to co-located apps [21]). 3. **Improper Configuration** refers to the configuration mechanisms that make the system vulnerable. These include Permission/Access Level (for File, Registry, etc) [12, 17, 19], Improper Server Configuration, Security Mechanism (Non-strict Security Mechanism in Firewall or Proxy), and Enable Specific Feature.
Figure 6: Identified Mechanisms during attack surface analysis

3.3.4 Network Level. mechanisms include (1) Accessible Private Network and (2) Using Unsafe Channel/Protocol [10, 12, 19] (using HTTP instead of HTTPS).

4 COMPARING TO RELATED WORK

The search strategy of our systematic literature review [34] consists of a manual search of four sources: the ACM Digital Library, IEEE Explore Library, ScienceDirect, and Springer Link. Our inclusion criteria are as follows: the work is (i) a full paper; and (ii) focus on discussing software system attack surface. Exclusion criteria are (i) position papers, short papers, keynotes, reviews, tutorial summaries, and panel discussions; (ii) not fully written in English; (iii) duplicated study; (iv) focused on attack surface outside the domain of software system; (v) focused on attack surface of a specific type of system (e.g., IOT), and (vi) used "attack surface" phrase instead of software vulnerability. We use the following search query: (Software OR Application) AND (Attack Surface OR Attack-Surface).

From our manual search, we collected a total of 2,150 papers. We applied our inclusion and exclusion criteria through reading the papers’ title, abstract and keywords (if present), resulting in 30 papers. Then, in this round we applied the inclusion and exclusion criteria by reading the full papers, resulting in a remaining 8 papers. These remaining papers were carefully reviewed, to verify the extent to which the findings from our study were supported by the literature or were complementary. Limited studies have been proposed for identifying attack surface of a software system.

Howard et al [12] described an approach for measuring the relative attack surfaces of two systems with regard to certain dimensions. This study measure relative attack surface in three abstract and limited dimensions which are targets and enablers, channels and protocols and access rights. Targets and enablers are resources that attacker can use such as process and data. For channels they considered two types of channels: message passing and shared memory. They also considered account, privilege level and trust-relationship as access rights. They added three attack vectors to the 17 attack vectors that Howard et al [11] identified for windows system. Authors argue that their approach does not attempt to provide a comprehensive way of measuring the attack surfaces, but rather provides a relative way for comparing two versions of a system. This study is system level because they considered the services that run on a system.

Similarly, Manadhata and wing in a TSE paper [17–19] proposed an attack surface metric to compare the security of two versions of a system to reason whether one is more secure than the other with respect to the attack surface metric. They proposed a model of system and its environment using state machine and considered any component that can be used to send/receive data to/from environment as an attack surface for Linux systems. They considered methods of the system, channels and data items as resources. They used the model to identify the accessible subset of resources (in terms of access rights) that contribute to the system’s attack surface. They defined that attack surface of the system in three dimensions:
data, channel, and system attackability (potential damage and the effort required to acquire an access right level). However, similar to prior work they did not aim for defining concrete and comprehensive attack surfaces. This study mainly focuses on metrics as approximations and showed the size of their metrics decreases when vulnerability is patched.

Huemann et al. [10] defined the components of the attack surface for web applications. They proposed attack surface of web application as a vector that has 22 dimensions that are categorized in 7 groups and considered weights for each components. They proposed Euclidean norm of the vector as attack surface indicator.

Nathan and Meeneely [25] proposed function and file level attack surface metrics. They considered entry and exit points and also dangerous system calls as attack surface components. They provided static and static+dynamic call graphs and calculated the proximity and risky walk metrics based on the call graph. They proposed three proximity-based metrics which are proximity to entry points, exist points and dangerous points.

Theisen et al. [32] performed a systematic literature review on attack surface definitions. They categorized the attack surface into 6 categories which are methods, adversaries, flows, features, barriers and reachable vulnerabilities. However, they mainly discussed granularity levels and concepts defined in previous works.

To evaluate the proposed attack surface categorization, we compare it with the attack surface components proposed in the literature. The comparison results (Table 1) show that the categorization provided by this paper covers all attack surface components introduced in the literature. The concepts which are missed in our categorization such as Search in Program and RPC and Named Pipe in Network entry points are specific concepts that can be covered by the proposed core categories. The comparison results indicate that the proposed attack surface model differs from the previously introduced components in that it:

- Provides a comprehensive attack surface categorization that considers different aspects in System, Network, Program, and Source levels. Previous works mostly focused on defining attack surface components by low level concepts.
- Defines clear concepts that can be part of an attack surface. For instance, Data Item which is defined as an attack surface component in [19] is a vague concept. Based on the examples mentioned for it, our categorization clearly indicates that the data item could be program or system data.
- Provides comprehensive Code Level attack surface components. For example, previous studies considered I/O methods as Entry Points [19, 25] at source level, however, we find that different Handlers and Indirect Entry Points can also be part of an attack surface. Nathan and Meeneely [25] defined System Calls as Dangerous Points, however we identify additional concepts as Dangerous Points such as Type Casting, Integer Operations, and Encryption/Decryption. We also define other code fragments such as Serialization/Deserialization, Reflection, etc. that can be target of attacks. Heumann et al. [10] define URL Parameters and Hidden Fields as input vectors, however some other important input vectors such as Post Request Parameter, HTTP Header, and Certificate identified in our GT analysis are missed.

We compare the low level concepts defined by our GT analysis with the concepts defined in the literature. The comparison results are shown in Table 2. The results indicate that the literature covers small percentages of Code level entry points, targets, and mechanisms which are in order 6.7%, 3.4%, and 3.3%. Network level mechanisms, System level entry points, and Program level mechanisms are major categories that are covered in the literature with 50%, 30% and 18.7%, respectively. In summary, the attack surface model proposed by this study is more comprehensive than the prior works, because a small percentage of the the attack surface concepts defined by our GT analysis is covered by the prior works in the literature.

5 DISCUSSION

Our analysis provides a very clear and comprehensive way to evaluate attack surface components at the System, Network, Program, and Source levels. While previous attack surface analysis works usually focus on specific types of systems like operating systems or web applications, this work defines a generic attack surface model that can be used to identify the attack surface components of different types of systems. Our work comprises attack surface components identified by earlier works and can be used to organize previously proposed concepts with more clear abstraction levels.

Attack surface components represent critical parts of a software system that could be targeted. Proposed attack surface model can be used as a check list to identify attack surface components in different software analysis approaches. For example, static analyzers or testers can identify system entry points and sinks for any type of system. The Source level Entry Points and Targets can be used for identifying sources and sinks during static and dynamic software analysis. Software analyzers and testers can use the defined concepts (like Direct Input and Indirect Input in the Program level of Entry Points for software symbolic execution. Proposed attack surface model can also be used by software developers to identify Targets and Entry Points in their systems and apply secure coding practices to secure vulnerable parts and/or prioritize security inspection efforts. As a summary, the proposed attack surface model can be used in different phases of software development life cycle (development, testing, maintenance) to improve the security of software systems.

6 CONCLUSIONS

Getting inspired from the similarity between the attack surfaces of a house/building and a software system, and asking three key questions (Where the attacks come from, What they target, and How they emerge), this paper develops a comprehensive attack surface model based on the Entry Points (Where), Targets (What), and Mechanisms (How) leveraged by cyberattackers. We follow a grounded theory-based approach to study attack surface components of software systems. Specifically, we focus on the software Entry Points, Targets and attack Mechanisms to define the attack surface components in our model. We find that there are four major theories for each of these three branches, i.e., Code, Program, System, and Network, and conduct a systematic literature review to verify to what extent previous studies corroborate with our findings. Preliminary results show that all attack surface components defined in the
Table 1: Comparison of the concepts in the proposed attack surface model with the literature

| Category | Level | N | NL | PL |
|----------|-------|---|----|----|
| Entry Points | C | 30 | 2 | 6.7 |
| | P | 42 | 2 | 4.8 |
| | S | 10 | 3 | 30 |
| | N | 20 | 3 | 15 |
| Targets | C | 59 | 2 | 3.4 |
| | P | 32 | 2 | 6.2 |
| | S | 23 | 2 | 8.7 |
| | N | 6 | 0 | 0 |
| Mechanisms | C | 30 | 1 | 3.3 |
| | P | 16 | 3 | 18.7 |
| | S | 10 | 1 | 10 |
| | N | 2 | 1 | 50 |

Table 2: Quantitative comparison of the concepts in the proposed attack surface model with the literature. N: number of concepts identified, NL and PL: number and percentage of concepts covered in literature, respectively

literature are covered by the proposed attack surface model, while prior works cover only a small portion of the concepts identified by our analysis. In best cases, they cover 50% of Network level, 30% of System level, 18.7% of Program level and 6.7% of the Code level concepts.

REFERENCES

[1] Cristian Cadar, Patrice Godfrey, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In 2011 33rd International Conference on Software Engineering (ICSE). IEEE, 1066–1071.
[2] Deng Chen, Yan-duo Zhang, Wei Wei, Shi-xun Wang, Ru-bing Huang, Xiao-lin Li, Bin-bin Qu, and Sheng Jiang. 2017. Efficient vulnerability detection based on an optimized rule-checking static analysis technique. Frontiers of Information Technology & Electronic Engineering 18, 3 (2017), 332–345.
[3] Juliet Corbin and Anselm Strauss. 2014. Basics of qualitative research: Techniques and procedures for developing grounded theory. Sage publications.
[4] CWE. 2021 (accessed August 14, 2021). CWE VIEW: Weaknesses Introduced During Design. https://cwe.mitre.org/data/definitions/701.html
[5] CWE VIEW: Weaknesses Introduced During Implementation. https://cwe.mitre.org/data/definitions/702.html
[6] B.G. Glaser. 1978. Theoretical Sensitivity: Advances in the Methodology of Grounded Theory. Sociology Press. https://books.google.com/books?id=73-4AAAAMAAJ
[7] Barney G Glaser. 1992. Basics of grounded theory analysis: Emergence vs forcing. Sociology press.
A Grounded Theory Based Approach to Characterize Software Attack Surfaces

[8] Barney G Glaser and Anselm L Strauss. 2017. Discovery of grounded theory: Strategies for qualitative research. Routledge.
[9] G Glaser Barney and L Strauss Anselm. 1967. The discovery of grounded theory: strategies for qualitative research. New York, Adline de Gruyter (1967).
[10] Thomas Heumann, Jörg Keller, and Sven Türpe. 2010. Quantifying the attack surface of a web application. Sicherheit 2010. Sicherheit, Schutz und Zuverlässigkeit (2010).
[11] Michael Howard. 2003. Fending off future attacks by reducing attack surface.
[12] Michael Howard, Jon Pincus, and Jeannette M Wing. 2005. Measuring relative system’s attack surfaces. In Computer security in the 21st century. Springer, 109–137.
[13] Renad Jovanovac, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In 2006 IEEE Symposium on Security and Privacy (S&P’06). IEEE, 6–pp.
[14] Zhen Li, Deqing Zou, Shouhui Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2018. Sysveer: A framework for using deep learning to detect software vulnerabilities. arXiv preprint arXiv:1807.06756 (2018).
[15] Zhen Li, Deqing Zou, Shouhui Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepercker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 (2018).
[16] Ingo Lütkebohle. 2021. Statistics Results. https://nvd.nist.gov/vuln/search?form_type=Basic&results_type=statistics&search_type=all. [Online; accessed 02-April-2021].
[17] Pratyusa Manadhata and Jeannette M Wing. 2004. Measuring a system’s attack surface. Technical Report. Carnegie-Mellon Univ Pittsburgh pa School of Computer Science.
[18] Pratyusa K Manadhata and Jeannette M Wing. 2010. An attack surface metric. IEEE Transactions on Software Engineering 37, 3 (2010), 371–386.
[19] Pratyusa K Manadhata and Jeannette M Wing. 2011. A formal model for a system’s attack surface. In Moving Target Defense. Springer, 1–28.
[20] MITRE. 2021 (accessed August 14, 2021). Common Vulnerabilities and Exposures. https://cve.mitre.org.
[21] MITRE. 2021 (accessed August 14, 2021). Common Weakness Enumeration. https://cwe.mitre.org/data/downloads.html
[22] MITRE. 2021 (accessed August 14, 2021). CVE-2016-10259. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2016-10259
[23] MITRE. 2021 (accessed August 14, 2021). CVE-2019-15336. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2019-15336
[24] MITRE. 2021 (accessed August 14, 2021). CVE-2020-5319 https://cve.mitre.org/cgi-bin/cvename.cgi?name=2020-5319
[25] Nathan Munuah and Andrew Meneely. 2016. Beyond the attack surface: Assessing security risk with random walks on call graphs. In Proceeding of the 2016 ACM Workshop on Software PROtection. 3–14.
[26] Bindu Madhavi Padmanabhan and Hye Beng Kuan Tan. 2016. Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis. IEEE Software 10, 2 (2016), 54–61.
[27] Kendra L Rieger. 2019. Discriminating among grounded theory approaches. Nursing Inquiry 26, 1 (2019), e12261.
[28] Leonardo Sousa, Anderson Oliveira, William Ozumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Balduino Fonseca, Roberto Oliveira, et al. 2018. Identifying design problems in the source code: A grounded theory. In Proceedings of the 40th International Conference on Software Engineering. 921–931.
[29] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In Proceedings of the 38th International Conference on Software Engineering. 120–131.
[30] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. 2015. Approximating attack surfaces with stack traces. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2. IEEE, 199–208.
[31] Christopher Theisen, Kim Herzig, Brendan Murphy, and Laurie Williams. 2017. Risk-based attack surface approximation: how much data is enough? In 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, 273–282.
[32] Christopher Theisen, Nathan Munuah, Mahran Al-Zyoud, Jeffrey C. Carver, Andrew Meneely, and Laurie Williams. 2018. Attack surface definitions: A systematic literature review. Information and Software Technology 104 (2018), 94–103. https://doi.org/10.1016/j.infsof.2018.07.008
[33] Yanzhi Wang, Dan Wang, Wenbing Zhao, and Yuan Liu. 2015. Detecting SQL vulnerability attack based on the dynamic and static analysis technology. In 2015 IEEE 39th Annual Computer Software and Applications Conference, Vol. 3. IEEE, 604–607.
[34] He Zhang, Muhammad Ali Babar, and Paolo Tell. 2011. Identifying relevant studies in software engineering. Information and Software Technology 53, 6 (2011), 625–637.