System-based Approach to Software Vulnerability

Sabah Al-Fedaghi
Computer Engineering Department
Kuwait University
Kuwait
sabah@alfedaghi.com

Abstract—The focus of vulnerability research has been conceptualization of the lifecycle of software vulnerability as errors in software that can be used by an attacker to gain access to a system or network. This lifecycle is described in terms of its phases: creation, discovery, exploitation, disclosure, patch availability, and patch installed. The objective of this paper is to clarify the notion of vulnerability so it complements current error-focused conceptualization. The paper proposes a fine-grained lifecycle of a vulnerable system in terms of a flowsystem that includes five basic stages and is defined by a flow transition diagram. A software system is first created, released, and transferred to users; it is then activated until it fails as a result of vulnerability to an attack. Several other phases lead to recreation of the system. Accordingly, vulnerability is defined as the state of a system where it can be damaged when it receives a certain type of attack.

Keywords—software vulnerability lifecycle, flow system, software error, risk.

I. INTRODUCTION

Security failures are often attributed to poor design or implementation. Understanding the notion of vulnerability is a key factor in security analysis of attacks. The typical approach to providing protection against vulnerability is to react to each vulnerability by applying patches to close “holes” in the software, thus countering any exploitation. Large databases of vulnerabilities disclosed in previous years are publicly available, for example, the National Vulnerability Database [18], the Open Source Vulnerability Database [19], and the CVE database [13].

Alternatively, instead of reacting to exploitation, protection can be based on “finding” vulnerabilities and offsetting possible multiple attacks. This involves preemptive protection by finding vulnerabilities with such techniques as examining the source code, and fuzz testing.

These approaches assume that vulnerability is caused by programming or design errors. They conceptualize the dynamics of the security system in terms of “vulnerability lifecycle.” For example, Frei et al. [14] conceptualize the lifecycle of vulnerability in terms of phases between distinctive events: creation, discovery, exploitation of availability, disclosure, patch availability, and patch installation, as shown in Fig. 1.

Vulnerabilities are typically created by accident as the result of a coding mistake, often involving mismanagement of memory [14]. This is a very restrictive view of vulnerability creation.

Early stages of a software development cycle involve requirement analysis, design, and implementation. A more comprehensive view of vulnerability creation may take into account that the origin of vulnerability can be found at any point in these stages. Specifically, specification of requirements may lack security constraints that cause vulnerability. In a voting system, the lack of explicit constraint of “one person, one vote” provides an opportunity for multiple votes by one person, even if design and implementation are error free. A more comprehensive approach is to release the creation of vulnerabilities from technical errors such as programming mistakes.

A conceptual blurriness appears when we ask, What is that thing created during the creation phase of vulnerability lifecycle? According to [14],

Defining vulnerabilities is a delicate undertaking that depends significantly on the parties involved and their intent. For example, whether a specific software flaw is considered a defect, a feature, or vulnerability differs whether you talk to a researcher, the vendor, or different users of the software.
Neverthelesss, many definitions of vulnerability have been given [16, 20]. Vulnerability has been defined as “a problem (such as a programming bug or common configuration) that allows a system to be attacked or broken into” [15]. Or, it is “the existence of a flaw or weakness in hardware or software that can be exploited resulting in a violation of an implicit or explicit security policy” [11]. A standard definition is that vulnerability is an error in software that can be utilized by an attacker to gain access to a system or network [17].

Suppose that software designers and programmers did a perfect job of turning requirements into a software system. In the example of a voting system, is the lack of explicit requirement constraining “one person, one vote” an error in the software? Suppose the system builder (e.g., a vendor) could not predict a new type of technology that makes it vulnerable to attack. Is this vulnerability an error? In these cases, an “error” is not a piece of program that can be fixed; rather, it is the product of requirement analysis, and its understanding rests on others being able to decompose and assess that analysis. Systems are vulnerable to failure, and errors are proxies for failures.

In addition, focusing on error leads to stop a causal analysis as soon as the error is discovered and corrected. In this case error is the end goal, not the starting point, putting immediate solution of the problem ahead of the aim of achieving a secure system. Alternatively, we propose a conceptualization of vulnerability as an evolving state of a system. Thus, the creation “event” in Fig. 1 is the creation of a vulnerable system, discovery is recognizing that a system state is a failure state resulting from an attack, etc. We can notice the heterogeneity of the events in Fig. 1 and the lack of systematization, where events start with vulnerability and end with what resembles a lifecycle of a patch.

Vulnerability can also be defined as a feature or characteristic of a system. For example, vulnerability is defined as the tendency of a system to be damaged [22]; however, a feature of a thing tends to characterize it over time. It is possible for a system to become vulnerable only because a new type of attack has been developed. An obvious example is progress in hardware such as increased CPU speed, and development in theoretical research resulting in an “old” system becoming vulnerable. For example, in an article titled “Graphics Processors to Speed Up Password Cracking,” it is reported [21].

Thanks to massive amount of parallel processing engines, graphics chips can speed up password cracking by a factor of up to 25…

With ElcomSoft’s new technology, the process [cracking the logon password for Windows Vista] would take only three to five days, depending upon the CPU and GPU.

Accordingly, vulnerability is not a (permanent) feature of a system. Alternatively, we define vulnerability as a “state” of a system, where it can be damaged by a certain type of attack.

The objective of this paper is to clarify the notion of vulnerability that complements current error-focused conceptualization. The next section is a summary description of the model that will utilized in achieving such an objective.

II. FLOW MODEL

The flow model (FM) is basically a lifecycle specification of things that flow (e.g., information). It has been used in several applications [1, 2, 3, 4, and 5]. This section reviews the basic model as it has been described in other publications, and it includes new aspects of the model such as triggering mechanism of the model. In the next section, we introduce the new idea that flow systems are flowthings.

A. General view

A flow model is a uniform method for representing things that “flow,” i.e., things that are exchanged, processed, created, transferred, and communicated. “Things that flow” include information, materials (e.g., manufacturing), and money.

To simplify this review of FM, we introduce the model in terms of information flow. There are five states of information: transferred, received, processed, created, and released, as illustrated in Fig. 2. The model can also be defined in terms of a transition graph with five states, as we describe later in the paper. Information can be stored, copied, destroyed, used, etc. in any of the five generic stages. As a flow structure, the model is characterized by five stages: receiving, processing, creating, releasing, and transferring. In Fig. 2, flows are denoted by solid arrows and may trigger other types of flow, denoted by dashed arrows.

The environment in which information exists is called its infosphere (e.g., computer, human mind, organization information system, department information system). The flowsystem is described in terms of a five-stage schema. Its transition diagram is reusable because a copy of it is assigned to each entity (e.g., software system, vendor, and user). An entity may have multiple flowsystem, each with its own flowsystem.

Identifying flowthings in conceptualizing a system is a fundamental first step in FM. Flowthings are things that can be received, processed, created, released, or transferred.

![Figure 2. State transition diagram for FM with possible triggering mechanisms. Solid arrows indicate flows, and dashed arrows represent triggerings.](image-url)
A flowsystem may not necessarily contain all stages, for example, conceptualization of a physical airport can model the flow of passengers: arriving (received), processed (e.g., passports examined), released (waiting to board), and transferred (to planes); however, airports do not create passengers (ignoring the possibility of an emergency where a baby is born in the airport). In this case, the flowsystem includes only the stages of received, processed, released, and transferred.

B. Exclusiveness of states

The states shown in Fig. 2 are exclusive in the sense that if information is in one state, it is not in any of the other four states. Consider a piece of information σ in the possession of a hospital. Then, σ is in the possession of the hospital and can be in only one of the following states:

1. σ has just been collected (received) from some source, patient, friend, agency, etc. and stored in the hospital record waiting to be used. It is received (row) information that has not been processed by the hospital.

2. σ has been processed in some way, converted to another form (e.g., digital), translated, compressed, etc. In addition, it may be stored in the hospital information system as processed data waiting for some use.

3. σ has actually been created in the hospital as the result of doctors’ diagnoses, lab tests, and so forth. Thus, σ is in the possession of the hospital as created data to be used.

4. σ is being released from the hospital infosphere. It is designated as released information ready for transfer. Analogous to a factory environment, σ represents materials designated as ready to ship outside the factory. It may actually be stored for some period waiting to be transported; nevertheless, its designation as “for export” keeps it in such a state.

5. σ is in a transferred state where it is being transferred between two infospheres. It has left the released state and will enter the received state, where it will become received information in the new infosphere.

It is not possible for processed information to directly become received information in the same flowsystem. Processed information can become received information in another flowsystem by first becoming released information, then transferred information, in order to arrive (be received) at the other.

Consider the seller and buyer information spheres shown in Fig. 3. Each contains two flowsystems: one for the flow of orders and the other for the flow of invoices. In the seller’s infosphere, processing of Order triggers (circle 3) the creation of Invoice in the seller’s infosphere, thus initiating the flow of invoices.

The reflexive arrow of the transfer state in Fig. 2 (state transition diagram) denotes the flow from transfer stage to transfer stage of another flowsystem.

In Fig. 3, the Buyer creates an Order that flows by being released and is then transferred to the Seller. The “transfer components” of the Buyer and the Seller can be viewed as their transmission subsystems, while the arrow between them represents the actual transmission channel.

C. Formal view

Fig. 3 illustrates the triggering mechanism between flows of orders and invoices. An important principle in FM is the separation of flows. Orders trigger invoices, and each has its flowsystem in their information sphere. Triggering in the context of FM means activation of a stage or substage, which may generate a flow. Suppose the receive stage is activated by triggering; when flow is received, triggering may then result in:

(1) Activating the flow to release
(2) Activating the flow to process
(3) Mistriggering

Mistriggering indicates that the triggering has not succeeded. Triggering may specify a chain of flow. For example, a triggering in receive may specify flow to release or flow to release and transfer. In the last case the triggering is a chain of triggering.

FM reflects a map of possible flows, just as a city map represents possible routes. Traffic lights internally trigger flows. Another example is illustrated in Fig. 4, where there are two possible triggers, A and B. A “causes” the creation of, say, a flowing and its flow to release and transfer stages. B “causes” receipt of a flowing and its flow to process, release, and transfer. The exact specification of any trigger is embedded in the trigger itself.
Fig. 5 shows FM with secondary stages. For example, \{Copy, Store, Delete, and Destroy\} can represent these secondary stages, as illustrated in the figure. Secondary stages are present in every primary stage. For example, there is stored received information, stored created information, stored processed information, stored released information, and stored transferred information.

One “inaccuracy” in FM is the arrows representing (Release, Receive), (Release, Process), and (Release, Create). Each arrow denotes a “return” flow. For example, if a communication channel is down for a long time, it may be decided to return the “message” to the sender (creator, processor, or receiver) who previously released it. For simplicity’s sake, our formalization does not guarantee that the released message is “returned” to its previous state, i.e., an internal sender. If the flowsystem represents a company, then receiving, processing, and creating information are handled by three different departments.

This formalization can be complemented by rules and constraints that permit flow from one state to another.

III. FLOWSYSTEMS AS FLOWTHINGS

Flowsystems are flowthings. For example, Microsoft as a flowsystem creates, releases, and transfers (software) flowsystems. This is illustrated in Fig. 6. Thus, flowsystems can be “born” by flowsystems. It is possible that a “born” flowsystem becomes a sibling of its originating system. For example, a department in an organization may give birth to a new department that was originally part of it. The “computer engineering subdiscipline” in the Electrical Engineering Department may become a department by itself within the College of Engineering. The new flowsystem has not been “transferred” to another flowsystem; rather, it has “delivered” to the global sphere of the parent (College of Engineering). It is possible that a flow system “transfers” itself to two new flowsystemss (e.g., an Electrical and Computer Engineering Department becomes two departments). The ECE department creates and delivers two new departments to the encompassing flowsystem (the college).

In addition, a system moves from one state to another according to flows of flowthings. We limit our interest to attacks only. An attack is a flowthing that moves a flowsystem to another state (like a transition function on state diagrams). We limit our interest to attacks that lead to a damaged (failed) state of flowsystems, including:

- Outside flow: Another flowsystem transfers attacks to the flowsystem.
- Interior flow: E.g., previously arrived attacks flow further into system stages.
- Interior triggering: E.g., attacks cause creation of new attacks.
- Exterior triggering: Another flowsystem triggers creation of attacks.

At certain point, an attack moves the flowsystem to a failed state as will be discussed later.

We will use the term “attack” to refer not only to the action itself, but also to the tool used in the attack (e.g., a virus). It can be created, received, processed, released, and transferred. An attack harms only systems that are vulnerable to that type of attack. It is possible that a flowsystem receives an attack, then neutralizes or destroys it, thus preventing it from proceeding to the processing stage. It is also possible that a flowsystem receives an attack that proceeds to the processing stage; however, the system prevents it from accessing created system files (e.g., a passwords file).
Consider a sample attack: a buffer overflow that occurs when data are written beyond the boundaries of fixed length buffers. This would be exploited by injecting malicious code, then executing a program that takes control of the system. In each step the system fails in correctly receiving inputs (failure 1), and incorrectly processing them (using them inside the program—exploitation and failure 2). Similarly, cross site scripting is a failure by the system at the receiving stage that may then be followed by another failure, where the system permits exploitation through bypassing access controls. Cross site scripting consists of the injection (receiving) of code in pages to be exploited by an attacker (processing stage).

In each case, if the attack is successful, it moves the flowsystem to a damaged state, as shown in Fig. 7. The term “damaged state” covers many types of malfunctions, including exploitation (by hackers), i.e., breakdown in system resilience that prevents it from protecting itself from misuse or abuse. Resilience in general refers to the ability of an entity to resist or recover [22]. Accordingly, vulnerability, as shown in Fig. 7, is a system state that moves to a damaged state by means of a successful attack.

IV. STATES OF A VULNERABLE SYSTEM

System can be viewed in terms of continuous levels of vulnerability gradation: a less vulnerable system, a more vulnerable system, ... an invulnerable system (e.g., a system with long-time resilience to attacks). A vulnerable system is a system with a risk of attack that results in a failure (damage). Risk is the likelihood of taking advantage of a vulnerable system and the impact of this vulnerability on the system that leads to damage.

In general, the vulnerability of a system can be identified in terms of tracing (a) flows and triggerings transferred to/from an adversary flowsystem, and (b) subcomponents of the flowsystem, as shown in Fig. 8. The open-headed twisted arrows denote these flows and triggerings.

In the next section we define the lifecycle of a vulnerable flowsystem that starts with creation of a flowsystem, until a damage state occurs because of a successful attack, result in turn in “re-creation” of a less vulnerable system, as shown in Fig. 9. In the figure, “creation of a new version of a vulnerable flowsystem” includes creating the system for the first time.

V. LIFECYCLE

We have already discussed the “vulnerability lifecycle” introduced by [14]. The error-based life cycle of vulnerability has been studied in many works (e.g., [7, 10]). In general it has the following stages [23]:

1. The creation of vulnerability.
2. The discovery of vulnerability.
3. The discovered vulnerability is disclosed.
4. The vulnerability is corrected.
5. The vulnerability is publicized.
6. The exploit is *scripted*.
7. The vulnerability *passes*.
8. The vulnerability *dies*.

On the other hand, a system-based lifecycle of a vulnerable system is shown in Fig. 10, starting with circle 1. Individual points in the lifecycle of a vulnerable system signify change in possible risks. A system is created, and then it is released. Releasing a system does not necessarily imply that it immediately goes to users. As in movie production, a product may stay in inventory waiting for a decision regarding opening day. In this rapid time progress, the length of this period between finishing a software product and its actual release to users may affect the risk of an attack that results in failure. “Attack technology” progress is like that of any technology; hence, release time ought to coincide with current state of art in that technology.

In Fig. 10, a system may be transferred to a user (circle 2), but stays in buffer waiting to actually “arrive” at the user. This phenomenon is analogous to an e-mail that reaches the e-mail server but is not opened. In the next phase, the user processes (circle 3) the product by activating it as one of the user’s processes. Thus, the system interacts with the “outside world,” repeatedly subjecting itself to attacks (circle 4), until a failure occurs (black flowsystem in Fig. 10). The rest of the lifecycle phases are shown in Fig. 10.

There are thirteen phases in the system lifecycle. The so-called discovery occurs when the system reaches a failed state, or it may occur sometime after that. The discovery point may not coincide with the failure point because of lack of immediate recognition that the failure is due to an attack, and not due, say, to ordinary errors.

In Fig. 10, after reaching a failed state, there is the release stage. Release can be considered as a discovery point, assuming an immediately action (the failed flowsystem is sent to vendor) is taken upon discovered that an attack is the cause of the system failure. Notice that several variation of this scenario does not affect the generality of this discussion, such as utilizing patches, shellcodes, and who exactly fixes the system (e.g., multiple vendors). A failed system plus a patch produces a new version of the system. It is possible that a report of the discovered attack is what actually sent since the vendor has a copy of the failed system.

Also, a variation of Fig. 10 can incorporate such actions as public description of the attack, and sending reports to experts.

An import point that is raised in Fig. 10 is that there a possible difference between the point of the effect of the attack (failure), and the discovery event. Also, there are possible difference regarding the events of disclosure, and correction. Fig. 10 shows systemic fine grained marking of points in the lifecycle of the vulnerable system. A mentioned previously, exploitation (by hackers) is a type of failure, i.e., breakdown in the resilience of the system that prevented it from protection itself from misuse or abuse.

Fig. 11 shows the circulation of the system in the 13 phases of its lifecycle.
VI. MODELING ATTACKS

To understand vulnerabilities, models are created to analyze conditions in the software lifecycle that lead to their appearance in the software system. This understanding is used to develop a system vulnerable to “known vulnerabilities.” For instance, the approach called Vulnerability Cause Graph (VCG) [8, 9] utilizes a directed acyclic graph that maps vulnerability to cause nodes. Each node represents a condition or event during the software development phase that might contribute to vulnerability. Utilizing such a graph, the software developer can study causes, scenarios, and actions that can lead to the vulnerability under study.

In VCG, a sequence of causes must be prevented to avoid vulnerability, the root node in the graph. We propose that understanding of certain vulnerabilities can be achieved by examining the flow of information during an attack. The information flow reflects the sequence of events that leads to system failure. The following example illustrates this flow-based method.

Consider the "SQL Injection" attack, described as follows.

The idea is to convince the application to run SQL code that was not intended…. So the first test in any SQL-ish form is to enter a single quote as part of the data: the intention is to see if they construct an SQL string literally without sanitizing. When submitting the form with a quote in the email address, we get a 500 error (server failure), and this suggests that the "broken" input is actually being parsed literally…

When we enter steve@unixwiz.net' the SQL parser finds the extra quote mark and aborts with a syntax error [12].

The attack is ended by accessing the file of the e-mails, including passwords. By drawing the flow of information involved in communication between the (supposed) hacker and the system, we can recognize conditions that lead to the appearance of vulnerability.

Fig. 12 shows a partial view of the scenario described completely in [12]. First, the receive stage of the system should set an alarm when input is the type of data that resembles an SQL-like string (circle A). At the processing stage (circle B), the sequence of processes to be designed with special security consideration involves:

1- Processes that inject statements into a program,
2- An SQL parser that creates errors, with careful consideration to reporting an error,
3- Activating SQL programs that:
   - Access a system file: A system file is created information that ought to have special security protection, compared with received or processed information.
   - Released information: The released stage ought to be in a special alert state when the released information is created information.
   - Transferred information: The transfer stage is the last line of defense to catch unauthorized output.

The spots in the stream of flow represent locations of necessary security checkpoints. Thus, multilevel checkpoints can be established in the developed system. These points and the type of checks may be used to guard against other types of attacks (basic operations). The method gives critical locations along with the stream of flow and identifies fine-grain operations that may cause a system alert.

![Figure 12. Modeling SQL injection attack.](image-url)
VII. CONCLUSION

This paper has introduced a methodology for conceptualizing the lifecycle of a vulnerable software system that complements the currently widely used method based on error-based descriptions of vulnerability. One disadvantage of over-focusing on error is that error has become the end goal, not the starting point, putting immediate solution of the problem ahead of the aim of achieving an invulnerable system. Alternatively, the focus on a system-based description is on the evolution of the software system as a vulnerable system. To give an illustration, the error-based approach to vulnerability is analogous to medical research that concentrates on collecting knowledge about diseases, including understanding them, their classification, and their treatments. In contrast, a system-based approach is analogous to patient-centered research where each patient is followed and analyzed through registering his/her cycles of sickness and recovery. Both approaches have advantages and could be adopted simultaneously.

The paper introduces a fine-grain lifecycle of a system defined in terms of 13 stages of a flowsystem. It starts with the creation of a system released to users and activated, until its evolution of the software system as a vulnerable system. To model malfunctioning of flows, stages, and substages of the flowsystem also serves for modeling specific vulnerability to understand and analyze conditions in the flowsystem. The flowsystem is defined as the state of a system that moves to a damaged state by an attack. A damaged state is malfunctioning of flows, stages, and substages of the flowsystem. The flowsystem also serves for modeling specific vulnerability to understand and analyze conditions in the software lifecycle that lead to failure upon receiving attacks.

A great deal of research can be taken in the context of this new approach to vulnerability. Data can be collected for the thirteen stages of the lifecycle of vulnerable systems in the same manner to current statistical analysis of error-based view of vulnerability. Many current known vulnerabilities can be modeled based on the flow model description.

REFERENCES


