Abstract

Preventing exploits from compromising software applications requires a fundamental understanding of how they are being exploited, and then leveraging that understanding in the formulation of tests that reveal software application vulnerabilities. To advance that understanding this paper first presents a Process/Object Model of Computation that establishes a relationship between software vulnerabilities, an executing process, and computer system resources such as memory, input/output, and cryptographic resources. That relationship promotes the concept that a software application is vulnerable to exploits when it violates (a) constraints imposed by computer system resources or (b) assumptions made about the usage of those resources. Secondly, the Process/Object Model also serves as a foundation for the definition of a Taxonomy of Vulnerabilities. That is, the computer system resources (or objects) identified in the Process/Object Model form the categories and refined subcategories of the taxonomy. Vulnerabilities, which are expressed in the form of constraints and assumptions, are classified within the Taxonomy according to these categories and subcategories. This Taxonomy of Vulnerabilities is novel and distinctively different from other taxonomies found in literature, and is also outlined in this paper.

1. Introduction

Software applications are used for a wide variety of purposes ranging from personal use, such as e-mail and document processing, to managing critical infrastructures, e.g., national power grids. Additionally, the computers on which such applications execute are pervasive, and through networking facilities, are more accessible today than ever before. Moreover, the advent of the Internet has dramatically increased interconnectivity among computers, making it possible to remotely access most of the computers connected to the Internet. Both the widespread use and increased accessibility make software applications an enticing and easy target for exploits. Numerous reported exploits targeting these applications provide ample evidence supporting this observation [8, 16].

Most of the exploits (or attacks) target software applications running on computer systems. These applications are compromised by exploiting vulnerabilities present in them. A vulnerability is defined as a state of the system from which it is possible to transition to an incorrect system state [6]. In other words, a vulnerability is a software system defect which, when exercised, can produce undesirable or incorrect behavior [21]. In contrast, an exploit is the process by which one or more vulnerabilities are exercised to attack a system.

Because vulnerabilities are central to exploiting a software application, one can prevent an exploit by identifying, and subsequently eliminating, vulnerabilities present in a software application. However, identifying if and which vulnerabilities are present is a difficult task. Factors contributing to this difficulty include:

1. The complexity of software applications: Modern software applications are often large, complex, and contain thousands of lines of code. Furthermore, application complexity increases with the number of services it uses which are provided by the other applications.
2. The number of potential vulnerabilities: Because numerous vulnerabilities exist, attempting to identify the specific one(s) present in a software application from a list of possibilities is impractical.
3. The complexity of vulnerabilities: Some vulnerabilities, such as those used in the “time of check, time of use” exploit, involve multiple software components interacting together to produce the vulnerable system state. This introduces additional layers of complexity.

Individually and collectively, these factors make identifying vulnerabilities a formidable task. However, understanding (a) the particulars of vulnerabilities, (b) how they are exploited, and (c) their relationship(s) to software applications and computer system resources can facilitate this identification. In this paper we introduce a Process/Object Model of Computation and a Taxonomy of Vulnerabilities that address (a)–(c) above. The
Process/Object Model of Computation establishes the relationship between vulnerabilities, software applications and computer system resources. This relationship captures the notion that vulnerabilities exist in a software application if it permits violation of constraints and assumptions. Furthermore, the Process/Object model identifies process (or system) objects that are targets for exploitation, e.g., memory, input/output (I/O) and cryptographic resources. In turn, these objects serve as a categorization basis for the Taxonomy of Vulnerabilities. This taxonomy provides the structure and context for classifying the constraints and assumptions that introduce vulnerabilities. As reflected in the Process/Object Model of Computation, the top-level categories of the taxonomy are: main memory, Input/Output (I/O), and cryptographic resources. Each of these categories is further divided into subcategories in which the constraints and assumptions are uniquely classified.

Taken together, the Process/Object Model and Taxonomy establish a framework for reasoning about how software applications are compromised:

- The Process/Object Model identifies the objects that are targets for an exploit;
- The Taxonomy links violable constraints and assumptions to those objects;
- A vulnerability exists if a software application (or process) violates one or more of the underlying constraints or assumptions;
- An exploit employs methods that rely on the presence of vulnerabilities to attack associated objects.

This conceptual understanding simplifies the formulation of tests that assist in detecting vulnerabilities, removing them, and subsequently, preventing exploits.

The remainder of the paper is organized as follows: Section 2 highlights related work focusing on notable existing taxonomies; Section 3 presents the Process Object Model of Computation; Section 4 provides an outline of the taxonomy; and finally, Section 5 presents the conclusions and outlines future work.

2. Related Work

Substantial research has focused on creating security taxonomies. Many of them center on different aspects of a security exploit: some classify vulnerabilities, some methods of attack, and others security exploits. An additional discriminating factor is the component(s) of a computer system that are targeted for attack, e.g., the operating system, application software, and interaction protocols. This section introduces several notable taxonomies that represent important steps in attempts to understand and structure the concepts underlying software security.

Integrity flaws: At the Lawrence Livermore Laboratory, Abbot et al. developed one of the first security taxonomies as a part of the RISOS (Research In Secure Operating Systems) project [1]. This taxonomy categorizes operating system integrity flaws into seven categories.

Security Errors: Bisbey and Hollingworth [4] developed a classification scheme for security errors as a part of the protection analysis project. The final report for that project, presents a scheme that classifies operating system security errors into ten categories.


Security Faults: In 1995 Aslam [17] presented a taxonomy of security faults. This taxonomy, which was developed to organize information being stored in a vulnerability database, consists of three top level categories: operational faults (or configuration errors), coding faults, and environment faults. Aslam provides a further decomposition for the coding and operational faults categories.

Objects and Attributes: Krsul [11] extends Aslam’s work by decomposing the environmental faults category into subcategories defined by environmental objects. Each entity therein is characterized as either containing or receiving information, and has a unique name and set of operations that can be performed on it. For example, “running program” is an object, and as such, has information attributes associated with it. Those attributes are defined as data components of the object, and form subsequent category levels of the taxonomy. Krsul asserts that programmers make either explicit or implicit assumptions about these attributes, which in turn, can lead to environmental vulnerabilities.

Although Krsul’s taxonomy is the most detailed of those included in this section, it has some shortcomings. First, it is difficult to distinguish between objects and their attributes because of the interpretation latitude permitted by the taxonomy. For example, an environment variable can be considered as an attribute of the executing program (an object), or it can be considered as an object by itself. This is a source of ambiguity in the taxonomy. Furthermore, Krsul fails to elaborate on how assumptions lead to vulnerabilities. Nonetheless, the significance of
Krsul’s taxonomy is that it provides substantial insight into what constitutes vulnerabilities by focusing attention on assumptions that are responsible for them. The other taxonomies outlined in this paper also exhibit ambiguity [4, 6, 12], that is, a single security flaw can be classified as member of multiple categories.

The Taxonomy of Vulnerabilities outlined in this paper, however, is distinctively different from those developed to date. It is grounded in a computational model that serves as a basis for the novel classification scheme used by the taxonomy. Moreover, the proposed taxonomy is designed to avoid ambiguity. As we demonstrate in the next Section, a single constraint or assumption can only be associated with one resource, which implies it can only be classified within a single category.

3. The Process/Object Model of Computation

This section presents the Process/Object Model of Computation which identifies potential objects targeted by security exploits, and also serves as a foundation for defining the Taxonomy of Vulnerabilities. Because the term “software application” is so ubiquitous, and because both the Process/Object Model of Computation and Taxonomy apply to “software applications”, we first define a “software application” in terms of its relationship with other computer system components.

When executing, software applications are viewed as (software) processes. The term software process, as used here, refers to an executing program. The Process/Object Model focuses on software processes that operate above the level of the operating system. Although these processes utilize hardware resources, they do so through the interface provided by the operating system.

With “software process” defined, it is now possible to provide the details of the Process/Object Model. As illustrated in Figure 3.2, the model takes a high-level view of a software process executing on the computer system. The process can be visualized as a black box entity that takes input, performs some operations, and produces an output. However, it does not execute in a vacuum. To perform its requisite operations, it requires resources such as (a) memory to store data, instructions and other execution specific parameters, (b) input/output (I/O) to receive and store input, store output, present output, and finally (c) cryptographic resources to store secrets, and ensure data integrity. A software process, through the operating system interface, uses one or more of these resources to carry out its requisite functions.

Modern computer systems can be visualized as a hierarchy of components. Figure 3.1 presents a rendering of this hierarchy as a layered pyramid. At the bottom of the pyramid are the hardware or physical devices that make up a computer system and include memory, hard disk, processor, buses, and so forth. The operating system, which manages the hardware devices and provides an interface to them, occupies the layer above hardware. Finally, software applications, which occupy the layer above the operating system, provide specific services to the computer system users and to other software applications.

Figure 3.1 Hierarchal view of computer system

Figure 3.2 High level view of software process executing on a computer system

A software process, however, utilizes resources according to certain rules and limitations, which are defined by the resources themselves or by the operating system. These rules and limitations manifest themselves in the form of constraints that a software process must adhere to in order to function correctly. For example, the

---

1 Please note that there exist other resources that a software process uses, such as the processor. However, the model only covers resources whose usage leads to vulnerabilities.

2 Please note that cryptographic algorithms and protocols are considered as resources. We contend that these algorithms and protocols should be treated as a resource and should not, under any circumstances, be developed by software developers.
system memory assumes a constraint that limits the total amount of memory available to a software process. In addition to such constraints, the software process itself makes assumptions about the resources it uses. For example, the process assumes that the amount of memory required for its proper operation is provided. The process has to ensure that the assumptions it is making about usage of the resources are valid, and hold under current computational demands.

These constraints and assumptions underlie vulnerabilities. The authors contend that a vulnerable state arises when a software process fails to enforce a constraint assumed by a resource and/or makes an incorrect assumption about its usage. In general, the resources and/or the underlying operating system enforce most of the constraints. For those that are not enforced at the resource or OS level, it is assumed that the software process will enforce or adhere to them. Failure to comply with a constraint and assumption creates a vulnerability, and subsequently, the possibility of an exploit.

At this point it is important to revisit the difference between a vulnerability and an exploit. A vulnerability is present whenever constraints assumed by the resources or the assumptions made about their usage are ignored by the software process. An exploit is realized when these violable constraints or assumptions are compromised, that is, employed in an attack. To further illustrate this difference, consider the example of buffer overflow. The system state can be described as a buffer that is allocated on the program stack with no bounds checking being performed on it. The program stack (part of main memory) assumes the following constraint on the software process: Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer. Neither the hardware nor the operating system enforces this constraint; in effect the software process is assumed to enforce it. However, suppose the process makes the assumption that the data being accepted by the process and stored in the buffer will always be smaller than or equal to the size of the buffer, and hence, ignores the constraint. Then, both the constraint assumed by the system memory and the assumption made by the software are violable. These violable constraints and the assumptions are the reason that this system state is vulnerable. Such states can lead to undesired and incorrect functionality when the process accepts and stores data that is larger than the buffer, thereby overwriting data that lie beyond the bounds of the buffer.

The Process/Object Model presented in this section provides a novel perspective on vulnerabilities. It can be used as a theoretical foundation for the study of an individual vulnerability, or vulnerabilities in general. The model also affords a number of other distinct advantages. In terms of developing test strategies for detecting vulnerabilities, focusing on constraints and assumptions as embodied by the model has a clear advantage – that is, they form the high-level goals of potential test strategies. In other words, the goal of a test strategy is to test if the software process allows the violation of a constraint or an assumption; a vulnerability exists if either can be violated.

4. Taxonomy of Vulnerabilities

This section outlines the Taxonomy of Vulnerabilities. The taxonomy provides a classification of violable constraints and assumptions. We identify the constraints and assumptions currently classified by the taxonomy by analyzing vulnerabilities which in turn, are identified by analyzing security exploits. Books [10, 20], mailing lists [8] and websites [16, 15] serve as sources for obtaining exploit summaries.

The Process/Object Model presented in the previous section presupposes that constraints and assumptions are bound to the resources, an insight used to derive the classification scheme embodied in the taxonomy. That is, the scheme uses resources as categories of the taxonomy. Effectively, the three top-level categories of our taxonomy are (1) Main Memory, (2) Input/Output, and (3) Cryptographic Resources (Figure 4.1).

Each top-level category of the taxonomy is divided into additional subcategories, which themselves are resources, but represent different components of each top-level category. Each component has its own distinctive set of characteristics and is used by the software process in a unique manner. Therefore, each component has different constraints and assumptions associated with it, which necessitates the inclusion of separate subcategories for them.

The three main categories of the taxonomy and their subcategories are:

- **Main Memory**: Main memory refers to the storage space that a software process uses to store input, output, execution specific parameters, instructions, and other objects necessary for the execution of the process. Main memory is divided into two subcategories: dynamic memory and static memory. Dynamic memory is defined to be the program stack and the heap, and is used by the software process to store process data such as local variables, function frames, environment variables and dynamically allocated variables. On the other hand, static memory is what the software process uses to store (initialized and un-initialized) global data.
- **Input/Output (I/O):** I/O corresponds to resources such as the file system, network interface, standard input interface and the standard output interface. A software process uses I/O resources to receive input, communicate output, and to store temporary and permanent data. Similar to main memory, I/O is divided into two subcategories: the network interface and the file system. **Network interface** refers to the interface that enables the software process to communicate with other computers over the network. This interface, provided by the operating system, is defined in terms of ports that a software process uses to send and receive data. **File system** refers to the hierarchy of connected directories containing metadata about files, and the files themselves. The software process uses the file system to receive input, store output, and store temporary and permanent data.

- **Cryptographic Resources:** Cryptographic resources refer to the algorithms and protocols contributed by the discipline of cryptography. These resources provide the software process with the capability of securing data and resources. This category is also divided into two subcategories: randomness resources and cryptographic algorithms and protocols. **Randomness resources** provide to the software process random number sequences – a key requirement for cryptographic algorithms and protocols. The other subcategory, **cryptographic algorithms and protocols**, refers to the algorithms and protocols that provide services of confidentiality, integrity, authentication and nonrepudiation to the software process. It includes a vast range of algorithms and protocols, such as DES, Kerberos, SHA1 and PKI.

The taxonomy classifies a constraint or an assumption as a member of one of the subcategories based on the resource with which it is associated. For example, an assumption such as the software process will be provided with the required dynamic memory necessary for its execution is categorized under the dynamic memory subcategory, as that resource is the one in question.

It is important to note that the taxonomy views resources (its categories and subcategories) from the same perspective as that of the software process. That is, the taxonomy is not concerned with the physical or the operating system view of resources, but instead, (a) how the software process is using those resources, and (b) how the attendant constraints and assumptions apply to the software process.

We have identified constraints and assumptions associated with all the subcategories of the taxonomy. However, only the dynamic memory subcategory has been sufficiently completed. We are currently in the process of completing the identification of the constraints and assumptions associated with the remaining subcategories.

The remainder of this section provides a more detailed description of the dynamic memory subcategory, and presents selected constraints and assumptions associated with it. It also presents a detailed example of a vulnerability resulting from a violable constraint associated with dynamic memory.

---

**Figure 4.1 Taxonomy of Vulnerabilities**

![Taxonomy of Vulnerabilities Diagram]

---

Proceedings of the 2nd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC'06)
0-7695-2539-3/06 $20.00 © 2006 IEEE
4.1 Dynamic Memory

Recall, dynamic memory is the part of main memory whose size changes as the process executes. It consists of two components: the program stack and the heap. The program (or execution) stack is a contiguous block of memory used by the operating system and the process. It stores process data in the form of local variables, execution-specific parameters, and so forth. The heap, on the other hand, is blocks of memory used to store dynamically allocated variables. For example, in the “C” language the heap stores variables that are allocated using the malloc() call.

Dynamic Memory

1. Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer.
   This constraint is violated if the software process allows data larger than the size of the buffer to be written to the buffer. Since the data is larger than the buffer, it will overwrite memory that lies beyond the bounds of the buffer.

2. An integer variable/expression used by the process as the index to a buffer must only hold values that allow it access to the memory locations assigned to the buffer.
   This constraint is violated if the process does not restrict the value of the integer variable being used as an index to the array. A hostile entity can provide the process with any integer value to use as an index, which, in turn, gives it access to memory locations beyond the bounds of the array.

Figure 4.2 Constraints and assumptions associated with dynamic memory subcategory.

Figure 4.2 presents two of the thirteen constraints and assumptions associated with the dynamic memory (please refer to Appendix A for a listing of all such constraints and assumptions), most of which are applicable to data held by variables stored in it. Failure to comply with any of the identified constraints and assumptions gives rise to a vulnerable condition. The following section presents an example that illustrates the exploitation of a dynamic memory vulnerability.

4.2 A Dynamic Memory Example

Item 2 in Figure 4.2 is used to provide a detailed example of a vulnerability. A potentially vulnerable state occurs when a software process accepts from a user two integer values – a position to store data and the value of the data itself. The software process stores the data in an array allocated on the program stack and uses the position value as an index to the array. If we assume that the process does not (or fails to) restrict the range of values for the index, then the user is free to enter any integer value he/she desires.

A vulnerability results from a software process allowing any value as the index for the array. Users can exploit this vulnerability by providing the software process with an index value greater than the size of the array, which permits them to write to a memory location beyond the array. Because users also provide data to be placed at the designated index value, they can write any value to any memory location that lies beyond the bounds of the array. Thus, users can overwrite the return value on the program stack with the value of their choice, which gives them the ability to execute any code they choose with the same privileges as that of the executing software process. Thus, by failing to ensure that the constraint is not violated, the software process creates a vulnerable system state.

5.0 Conclusion and Future Work

This paper presents a Process/Object Model of Computing, which establishes the relationship between vulnerabilities, software applications, and computer system resources. This relationship captures the notion that software applications are susceptible to exploitation when they permit violation of constraints imposed by computer system resources or assumptions made about the usage of these resources. The model serves as a foundation for the Taxonomy of Vulnerabilities, which classifies these violable constraints and assumptions. Both the model and taxonomy reflect a classification (or categorization) scheme based on the resources (or objects) used by computational processes. In particular, the taxonomy has three top-level categories: main memory, input/output, and cryptographic resources. These are consistent with the objects identified by the Process/Object Model. Those categories are further divided into refined subcategories, and serve as the final classification categories for the constraints and assumptions.

We have completed the identification of constraints and assumptions associated with the dynamic memory subcategory, and are in the process of completing the remaining subcategories. We are also working on developing a process for adding new constraints and assumptions to the taxonomy as they are discovered. Such a process is necessary to ensure the longevity and usefulness of the taxonomy. We also realize the importance of establishing the effectiveness of the taxonomy relative to real world exploits. To this end, we are currently classifying vulnerabilities derived from
advisories issued by CERT [9]. Finally, we intend to use the taxonomy to develop an approach for assessing the security of software applications by using the identified constraints and assumptions, and their associated objects, to derive tests to test for the presence of vulnerabilities.

The Process/Object Model of Computing presented in this paper offers a new perspective on vulnerabilities. It establishes a relationship between resources, and constraints and assumptions that advances our understanding of how software systems are compromised. Furthermore, it serves as the foundation for a Taxonomy of Vulnerabilities. This taxonomy is both novel and distinctively different from others found in current literature.

References


### Appendix A: Constraints and Assumptions Associated With Dynamic Memory

#### Dynamic Memory

1. **Data accepted as input by the process and assigned to a buffer must occupy and modify only specific locations allocated to the buffer.**
   
   This constraint is violated if the software process allows data larger than the size of the buffer to be written to the buffer. Since the data is larger than the buffer, it will overwrite memory that lies beyond the bounds of the buffer.

2. **The process will not interpret data present on the dynamic memory as executable code.**
   
   This assumption is violated if the process is made to interpret data present on the dynamic memory as executable code, which, in turn, can be accomplished by changing process variables it holds, such as, return addresses, exception pointers.

3. **Environment variables being used by the process have expected format and values.**
   
   This is a violable assumption, because a hostile entity can change the environment variables before the process begins execution. These variables are provided by the operating system and define the behavior of the process. A violation can occur if the process uses these variables and makes assumptions regarding their format and values.

4. **The process will be provided with the dynamic memory that it requests.**
   
   This is a violable assumption, because the amount of dynamic memory available to a process is limited and depends on the total amount of memory available on the computer system and the number of running processes. A violation can occur if the process assumes that it has access to an unlimited amount of memory.

5. **Data present on the dynamic memory cannot be observed while the process is in execution.**
   
   This is a violable assumption, because a hostile entity can run the process in a controlled environment and observe the contents of the dynamic memory, including any privileged data it holds.

6. **Data owned by the process and stored on the dynamic memory cannot be accessed after the process frees the memory.**
   
   This is a violable assumption, because the memory being used by the process is not erased after the process frees it. Hence, another process, if allocated the same physical memory, can access the data left over by the previous process.

7. **A pointer variable being used by the process references a legal memory location.**
   
   This is a violable assumption, because a pointer variable can point to any memory location, including memory locations outside the process address space. Furthermore, it can reference wrong variables, thereby creating illegal memory references.

8. **A memory pointer returned by the underlying operating system does not point to zero bytes of memory.**
   
   This is a violable assumption, because some operating system’s can provide the process with a pointer that points to zero bytes of memory. Using this pointer will cause illegal memory references or overwriting of memory locations being used by other variables.

9. **A pointer variable being used by the process cannot reference itself.**
   
   This assumption is violated if a pointer variable references itself. The consequences of this violation vary from garbage value being written to the memory location to the process going into an infinite loop.

10. **Data accepted by the process must not be interpreted as a format string by the I/O routines.**
    
    This constraint is violated if a process accepts input and interprets it as a format string. A violation will at the very least reveal the contents of the process stack. Additionally, a hostile entity can provide the process a specially crafted format string that allows it to write data to the process stack.

11. **The value of an integer variable/expression (signed & unsigned) accepted/calculated by the process cannot be greater (less) than the maximum (minimum) value that can be stored in the integer variable.**
    
    This is a violable assumption, because the maximum (minimum) value of an integer variable is determined by the amount of storage space provided to it by the underlying operating system. If the process tries to store a value that requires more storage space than is allocated to an integer, then the higher order bits of this value are dropped, resulting in a wrong value being stored.

12. **An integer variable/expression used by the process as the index to a buffer must only hold values that allow it access to the memory locations assigned to the buffer.**
    
    This constraint is violated if the process does not restrict the value of the integer variable being used as an index to the array. A hostile entity can provide the process with any integer value to use as an index, which, in turn, gives it access to memory locations beyond the bounds of the array.

13. **An integer variable/expression used by the process to indicate length/quantity of any object must not hold negative values.**
    
    This constraint is violated if the process uses an integer value to indicate length/quantity of an object and does not restrict the value to only positive values. A hostile entity can use a negative value to indicate the length of the object, thereby creating an error condition.