19 Attacks for Exploiting (all) Applications

Dr. James A. Whittaker, Ph. D
Dr. Herbert H. Thompson, Ph. D
Security Innovation, Inc.

19-Attacks Overview. Security Innovation's testing techniques are not "black magic" like some other security service providers - we publish our security testing methodology and refine it with every project. Our 19 attacks, described in an abridged format in this whitepaper, are practical, effective and based on years of research into software security defects. These attacks are effective on any kind of application, platform and development language and are used by thousands of developers and testers world-wide. They are also the cornerstone of Security Innovation's methodology that subsumes all other security standards, including the OWASP top ten, and we pass them along to you for use in your security testing efforts.

Following is an overview of the 19 attacks; each is described in further detail later in the whitepaper, and each is described in extensive detail in our top-selling book *How to Break Software Security.*

I. Attacking Software Dependencies

Attack 1: Block access to libraries
Attack 2: Manipulate the application's registry values.
Attack 3: Force the application to use corrupt files
Attack 4: Manipulate and replace files that the application creates, reads, writes & executes
Attack 5: Force application to operate in low memory, disk-space and network-availability conditions

II. Breaking Security Through the User Interface

Attack 6: Overflow input buffers
Attack 7: Examine all common switches and options
Attack 8: Explore escape characters, character sets, and commands

III. Attacking Design

Attack 9: Try common default and test account names and passwords
Attack 10: Use Holodeck to expose unprotected test APIs
Attack 11: Connect to all ports
Attack 12: Fake the source of data
Attack 13: Create loop conditions that interpret script, code, or other user-supplied logic
Attack 14: Use alternate routes to accomplish the same task
Attack 15: Force the system to reset values

IV. Attacking Implementation

Attack 16: Get between time of check and time of use
Attack 17: Create files with the same name as files protected with a higher classification
Attack 18: Force all error messages
Attack 19: Use Holodeck to look for temporary files and screen their contents for sensitive information

I. Software Dependencies Overview

Applications rely heavily on their environment in order to work properly. They depend on the OS to provide resources like memory and disk space; they rely on the file system to read and write data; they use structures such as the Windows Registry to store and retrieve information; the list goes on and on. These resources all provide input to the software— not as overtly as the human user does—but input nonetheless. Like any input, if the software receives a value outside of its expected range, it can fail. Inducing failure scenarios can allow us to watch an application in its unintended environment and expose critical vulnerabilities.

Attack 1: Block access to libraries.

Software depends on libraries from the operating system, third-party vendors, and components bundled with the application. The types of libraries to target depends on your application. Sometimes DLLs have obscure names that give little clue to what they're used for. Others can give you hints to what services they perform for the application. This attack is designed to ensure that the application under test does not behave insecurely if software libraries fail to load. When environmental failures occur, application error handlers get executed. However, sometimes these situations are not considered during application design, and the result is the dreaded unhandled exception. Even if there is code to handle these types of errors, this code is a fertile breeding ground for bugs, because it is likely that it was subjected to far less testing than the rest of the application.

Attack 2: Manipulate the application's registry values.

The Windows registry contains information crucial to the normal operation of the operating system and installed applications. For the OS, the registry keeps track of information like key file locations, directory structure, execution paths, and library version numbers. Applications rely on this and other information stored in the registry to work properly. However, not all information stored in the registry is secured from either users or other installed applications. This attack tests that applications do not store sensitive information in the registry or trust the registry to always behave predictably.

Attack 3: Force the application to use corrupt files

Software can only do so much before it needs to store or read persistent data. Data is the fuel that drives an application, so sooner or later all applications will have to interact with the file system. Corrupt files or file names are like putting sugar in your car's gas tank; if you don't catch it before you start the car, the damage may be unavoidable. This attack determines if applications can handle bad data gracefully, without exposing sensitive information or allowing insecure behavior. A large application may read from and write to hundreds of files in the process of carrying out its prescribed tasks. Every file that an application reads provides input; any of these files can be a potential point of failure and thus a good starting point for an attack. Particularly interesting files to check are those that are used exclusively by the application and not intended for the user to read or alter; they are files where it is least likely that appropriate checks on data integrity will be implemented.

Attack 4: Manipulate and replace files that the application creates, reads from, writes to, or executes

Similar to Attack 3, this attack also involves file-system dependencies. In previous attacks, we were trying to get the application to process corrupt data. In this one, we manipulate data, executables, or libraries in ways that force the application to behave insecurely. This attack can be applied any time an application reads or writes to the file system, launches another executable, or accesses functionality from a library. The goal of this attack is to test whether the application allows us to do something we shouldn't be able to do.

Attack 5: Force the application to operate in low memory, disk-space and network-availability conditions

An application is a set of instructions for computer hardware to execute. First, the computer will load the application into memory and then give the application additional memory in which to store and manipulate its internal data. Memory is only temporary, though; to really be useful, an application needs to store persistent data. That's where the file system comes in, and with it, the need for disk space. Without sufficient memory disk space, most applications will not be able to perform their intended function. The objective of this attack is to deprive the application of any of these resources so testers can understand how robust and secure their application is under stress. The decision regarding which failure scenarios to try (and when) can only be determined on a case-by-case basis. A general rule of thumb is to block a resource when an application seems most in need of it.

II. User Interface Overview

The user interface is usually the most comfortable bug hunting ground for security testing. It's the way we are accustomed to interacting with our applications, and the way application developers expect us to. The attacks discussed here focus on inputs applied to software through its user interface. Most security bugs result from additional, unintended, and undocumented user behavior. From the UI, this amounts to handling unexpected input from the user in a way that compromises the application, the system it runs on, or its data. The result could be privilege escalation (a normal user acquiring administrative rights) or allowing secret information to be viewed by an unauthorized user.

Attack 6: Overflow input buffers

Buffer overflows are by far the most notorious security problems in software. They occur when applications fail to properly constrain input length. Some buffer overflows don't present much of a security threat. Others, however, can allow hackers to take control of a system by sending a well-crafted string to the application. This second type is referred to by the industry as "exploitable," because parts of the string may get executed if they are interpreted as code. What sometimes happens is that a fixed amount of memory is allocated to hold user input. If developers then fail to constrain the length of the input strings entered by the user, data can overwrite application instructions, allowing the user to execute arbitrary code and gain access to the machine.

Attack 7: Examine all common switches and options

Some applications are tolerant to varying user input under a default configuration. Most default configurations are chosen by the application developers, and most tests are executed under these conditions, especially if options are obscure or are entered using command-line switches. When these configurations are changed, the software is often forced to use code paths that may be severely under-tested and thus results can be unpredictable. Obviously, to test a wide range of inputs under every possible set of configurations is impossible for large applications; instead, this attack focuses on some of the more obscure configurations, such as those in which switches are set through the command line at startup.

Attack 8: Explore escape characters, character sets, and commands

Some applications may treat certain characters as equivalent when they are part of a string. For most purposes, a string with the letter a in a certain position is not likely to be processed any differently from a similar string with the letter z in that same position. With this in mind, the question "Which characters or combinations of characters are treated differently?" naturally follows. This is the driving question behind this attack. By forcing the application to process special characters and commands, we can sometimes force it to behave in ways its designers

did not intend. Factors that affect which characters and commands might be interpreted differently include the language the application was written in, the libraries that user data is passed through, and specific words and strings reserved by the underlying operating system.

III. Design Overview

It is very difficult to look at 300 pages of design documentation and determine whether the finished product will be secure. It is no surprise then that some security vulnerabilities creep in at the design phase. The problem is that subtle design decisions can lead to component interaction and inherent flaws that create vulnerabilities in the finished product. Attacks 6-8 help expose these design insecurities in software it will tackle insecure defaults, test accounts, test instrumentation, open ports, and poor constraints on user-supplied program logic.

Attack 9: Try common default and test account names and passwords

In applications that have restrictions on data and functionality, all users are not treated equally. User actions are governed by their assigned level of access. In most instances, users are identified and authenticated with user names and passwords. Many applications, however, ship with some special user accounts built in; the most common examples being the "Administrator" or "root" accounts. These accounts usually don't present a problem; they are typically well documented, and the user is prompted to change or initialize the password upon installation. Problems arise, though, when undocumented, invisible, or unconfigurable accounts ship with the product. We must understand when user credentials are entered, checked and whether or not they get cached. This attack is designed to weed out "hidden" accounts so that they can be dealt with before release.

Attack 10: Use Holodeck to expose unprotected test APIs

Complex, large-scale applications are often difficult to test effectively by relying on the APIs intended for normal users alone. Sometimes there are multiple builds in a single week, each of which has to go through a suite verification tests. To meet this demand, many developers include hooks that are used by custom test harnesses. These hooks—and corresponding test APIs—often bypass normal security checks done by the application the sake of ease of use and efficiency. Developers add them for testers, intending to remove them before the software is released. The problem is that these test APIs become so integrated into the code and the testing process that when the time comes for the software to be released, managers are reluctant to remove them for fear of "destabilizing" the code, potentially causing a major delay in the ship date. It is thus critical to find these dangerous programmatic interfaces and ensure that, if they were to be accessed in the field by hackers, they could not be used to compromise application, its host system, or user data. By watching the application with Holodeck while automated test suites are running, we can identify dlls that are loaded and used, and then evaluate their impact on application security. In this attack, we try to expose test APIs.

Attack 11: Connect to all ports

A port is a method of organizing network traffic that is received or sent from a machine, so that different types of data can be transmitted simultaneously. When a port is "open," the operating system is "listening" for data through that interface. Applications commonly open ports to send data across the network. However, an open port is not automatically a secure conduit for communication. Without proper measures taken by application developers, an open port is a welcome mat for a hacker attempting to gain unauthorized access to a system. This attack finds these "open doors."

Attack 12: Fake the source of data

Some data is trusted implicitly, based on its source; for example, applications tend to accept values from sources like the OS with minimal scrutiny. Some sources should be trusted (in fact they must be trusted) for the application to function—sources such as configuration commands from an authenticated administrator. Problems arise when the trust an application extends to a particular source is not commensurate with the checks it makes to ensure that data is indeed from that source. This attack focuses on ensuring that applications take the proper precautions to verify the source of data, and that even when verified, the level of trust the application extends to that source is appropriate.

Attack 13: Create loop conditions in any application that interprets script, code, or other user-supplied logic

Some commands, when executed in isolation, are harmless. Imagine opening a Web browser and navigating to a Website that launches another window. This action by itself may be annoying, but it doesn't represent a threat to the end user. Now imagine that new window launching a third window, followed by another and another. Without some sanity checks by the browser, the system would become deadlocked. This attack investigates these repeated actions: taking commands that are relatively benign and executing them over and over again to deny functionality to entitled users or processes.

Attack 14: Use alternate routes to accomplish the same task

How many ways can you open a Microsoft Word® file in Windows. You could:

- Type the path in the Run dialog box
- Double-click on the file's icon in an Explorer window
- Type the path and file name in an Explorer window
- Type the path and file name in an Internet Explorer window
- Select it from the My Recent Documents tab on the Start menu
- Type the file name in the Open dialog box within Word
- Etc., etc., etc.

There are many ways in which to accomplish this task, but whichever method we choose, we expect the same result: our document will be displayed and be ready to edit. Now imagine trying to implement a security control on that document. To be effective, we must anticipate every possible scenario and verify that each goes through our validation routine; this can be a daunting task for even the most experienced developer. Cases get missed, and the result is a route that circumvents security controls. This attack is designed to help you think about the applications you test and explore all possible ways of accomplishing a task, not just typical user scenarios.

Attack 15: Force the system to reset values

This is one of our favorites, because you don't really need to do anything; indeed, that's the whole point of the attack. Leave fields blank, click Finish instead of Next or just delete values. These types of actions force the application to provide a value where you haven't. Establishing default values is a fairly intricate programming task. Developers have to make sure that variables are initialized before a loop is entered or before a function call is made. If this isn't done, then an internal variable might get used without being initialized. The result is often catastrophic. Whenever a variable is used, it must first be assigned a legitimate value. Good programming practice is to assign a value to a variable as soon as it is declared in order to avoid these types of failures. In practice, though, programmers often assume that the user will provide a legitimate value in the course of the application's execution before the variable is used. For security, the biggest concern is that default values and configurations can leave the

software in an unsafe state. This attack is focused on forcing the application to use these default values and then assessing the vulnerabilities that these values produce.

IV. Implementation Overview

A perfect design can still be made vulnerable by imperfect implementation. For example, the Kerberos authentication scheme is renowned as a well thought out and secure authentication scheme, yet the MIT implementation has had many serious security vulnerabilities in its implementation, most notably, buffer overruns. Indeed, we can ensure that every aspect of the design is secure and still produce an unsecure product. The problem is that security is not communicated well down the design hierarchy. Developers are usually given a list of requirements that emphasize which interfaces their component should extend to the rest of the application and the form of data that their component will receive. They are also given requirements for the computation to be performed on that data. However, specific requirements are seldom given as to exactly how that computation should be performed.

Attack 16: Get between time of check and time of use

Data is at risk whenever an attacker can separate the functions that check security around a feature or a piece of data from the functions that actually access and use these features or data. The ideal situation would be to ensure that every time sensitive operations are performed, checks are made to guarantee that they will succeed securely. If too much time elapses between the time the data is checked and the time it is used, then the possibility of the attacker infiltrating such a transaction must be considered. It is the old “bait and switch” con applied to computing: Bait the application with legitimate information, and then switch that information with illegitimate data before it notices. This attack is designed to exploit this time delay and penetrate the process between these two functions, with the goal being to force the application to perform some unauthorized action.

Attack 17: Create files with same name as files protected with a higher classification

Some files enjoy special privileges based on their location. For example, take dynamic link libraries (dlls). These libraries are used to perform certain tasks and are loaded by the application either at startup or when needed. Depending on where these libraries are located in the directory structure, a user with restricted privileges may not be allowed to alter them or write to the directory that contains them. Attackers can take advantage of the fact that these libraries are usually loaded by name, without any further checks to make sure that they are indeed the desired files. This can be exploited by creating a file with the same name and placing it in a directory the user does have access to that the application may search first. A related issue is that some files are given special privileges based solely on their names. This is a common phenomenon, especially with antivirus software that operates using a complex mesh of filtering rules based on filenames and their extensions. This attack will target both behaviors and is applicable any time an application makes execution or privilege decisions based on filename.

Attack 18: Force all error messages

Error messages are used to convey information to a user. Their purpose is to alert a user to some improper or disallowed action that may have been attempted. Our goal in this attack is to try to force the range of error messages that the application can display. Those of you who read the original How to Break Software book may remember this attack. In that book, our focus was to think through the possible illegal values that could be entered into a field. By trying to cause error messages, you are actually covering the range of bad input to the application and testing its robustness to that data; an example would be trying to enter a negative value into a “number of siblings” Web field. Obviously, this is an illegal value, but by

trying to force the application to display an error message indicating this, we are actually testing the application's ability to appropriately handle illegal input. The goal of this attack is to find a situation that is not handled appropriately; that is, no error message is displayed, and the application attempts to process the bad value.

Attack 19: Use Holodeck to look for temporary files and screen their contents for sensitive information

Applications routinely write data to the file system, which can store both persistent (permanent) data and also temporary data. Temporary files can be created to transfer data between components or to hold data that may be either too large to hold in memory or too inefficient to keep there. If this data (CD keys, encryption keys, passwords, or other personal data) is sensitive, then the mechanism for storing this data and access points to this data need to be investigated. Testers, especially security testers, must be aware of when, where, and how the application accesses file-system data. To be effective, we must identify which data should not be exposed to other potential users of the system. After sensitive data has been identified, we must find creative ways to gain insecure access to it. It is important to keep in mind that software lives in a multiuser, networked world, and just because our software created a file and deleted it doesn't mean that those operations went unnoticed by prying third parties. Our primary observation tool for this attack is Holodeck, which can monitor the application for file-writes and log these behaviors so that we can sift through them. Holodeck can tell us the when, how and where of every file access an application makes. It's up to us to investigate what was written and whether this data is supposed to be secret.