Massachusetts Institute of Technology
6.857: Network and Computer Security
Professor Ronald L. Rivest

Handout 13
October 5, 1999

Problem Set 2 Solutions

Problem 2-1. Trusting Trust (Nick Estrada, Nick Feamster, Michael Freedman)

Ken Thompson's article "Reflections on Trusting Trust" addresses the possibility of generating a bogus compiler, such that a C program which a user writes to perform some actions might in fact perform additional actions (or completely different actions altogether) than intended. Additionally, the compiler is able to "cover its tracks", so that, upon casual observation, the user cannot detect that any suspicious activity is taking place.

Thompson's article deals with an attack which results from trusting code which the user did not write himself (i.e., the code which compiles the programmer's C code). Thompson argues more generally that this concept can be applied at any level (e.g., the assembler, the loader, hardware microcode, etc.).

To the extent which Thompson generally argues that such an attack can occur at any level, Java does not offer any solution to this problem. A well-installed microcode bug can occur as a result of running a Java VM as easily as it can result from compiling an executable from a "buggy" C compiler. Additionally, because the architecture of Java involves many more components (i.e, the bytecode verifier, the classloader, the VM, the security manager, etc.), many targets for attack are available. An attacker need only compromise one of these components to create a Trojan horse.

Applet Security in JDK 1.2: Overview

Java Applet security is largely enforced by applet class loaders. Class loaders first enforce the Java name space hierarchy, then run classes through the verifier, which ensures four things: no stack overflows or underflows occur, validity of register accesses and stores, correct bytecode instruction parameters, and no illegal data type conversion. In JDK 1.2, all bytecode, regardless of locality, is subject to a security policy. This is an improvement over JDK 1.1.x, which automatically trusts local or digitally signed applets. The security policy creates various permission domains, ranging from full access (the JDK 1.x answer for local files) to completely encapsulated sandboxes (the JDK 1.0 answer for remote code.)

The Java security model contains vulnerabilities which can be exploited such that the programmer or the user downloading an applet is unaware of the exploit. There are vulnerabilities which can potentially be exploited in the following areas:

- **The Security Manager/Class Loader:** through the generation of bogus security policies, falsified security domains, and non-local applets executing on the local machine without the user's knowledge.
- **The Bytecode Verifier:** through clever use of native libraries and the Java Native Interface.
- **The Virtual Machine:** through distribution of a VM with Trojan horses.

Exploiting the Java 2 Security Manager

The Policy Tool of Java 2 has users add policy entries that grant directories permission to perform certain actions. These permissions may include the ability to read/write files, load libraries on the client, execute local processes, exit the Java VM, or bypass bytecode verification.

Possible Exploits

Even with the additional security that Java 2 offers, hostile applets may be able to bypass restrictions and gain control of user systems by forging security domains, compromising the security manager, and using local applets to alter files such as the security policy.

- **Accessing Domains:** If an applet can insert code into "trusted" directories, the code can exploit the directory domain to escape necessary class loading and/or bytecode verification. One such possibility resides in the browser's or appletviewer's cache, which is sometimes granted system-wide permissions. Therefore, if the downloaded, cached applet can detect or determine its locally cached name, it may execute its hostile code within the cache's security domain. In the JDK 1.x scheme, files residing on the local disk and in directories included in the CLASSPATH have full access. The CLASSPATH can include the cache, resulting in the described security hole.

- **Class Loader and Security Manager:** The class loader and security manager are established at startup and cannot be extended, overloaded, overridden or replaced, or be referenced or created by an applet. However, if we have an exploitable domain, we might be able to insert or modify these files. Although these changes will not take effect until the next startup, we can thereafter trick the browser into running a compromised class loader or security manager.
- **Local Applets:** A local applet could conceivably be written to read and write files, given that the `acl.read` and `acl.write` function are set to the directories I wish to write to and read from, respectively. Using such an applet (and assuming that we're looking at the appropriate directory), it is possible to write a bogus security file with the applet and subsequently load an `AccessController` which refers to this new bogus security policy. Using such a scheme, it might be possible to grant myself permissions with a new security policy, and subsequently remove the policy file I created at the completion of the applet. It is also likely that this operation could be performed with a native library exploit, as described in the "Java Native Interface" section.

Problem 2-2. Trusted Signatures: The security policy model results in certain signers or authorities to be granted various permissions on the client. If these signers were to succumb to attack, the security breach would be passed on to the trusting client. For example, an attacker manages to insert something into a directory that is packaged into a archive, signed, and shipped. Then, the trusting the client will implicitly trust this JAR file, and thus permit compromised code to execute on its machine.

Covering Tracks

Thompson's "buggy" C compiler `cc`'d its source to binary code, then covered its tracks by copying "unbugged" source back into the directory. The binary is bugged, the source appears clean. Likewise, this type of attack can be extended into the Java 2 realm as well. Not only are we trying to exploit a user; we do not want the user to realize that he has been exploited.

- **Primordial Class Loader:** The class resolution algorithm in JDK 1.2 delegates responsibility of class loading to a "primordial" class loader, if the class has 1) not already been loaded and 2) a delegation parent for the current class loader does not exist. Primordial class loaders are written in a native language, such as C. As such, they are exploitable per methods described in the native language section.
- **java.security.policy:** After a rogue applet has presumably written a temporary bogus security policy file (to be deleted at a later point by the same applet, or maybe a cron job), the VM could conceivably load this new policy file without the user knowing even about its existence by surreptitiously setting the `java.security.policy` variable at runtime to be the bogus file. The file could then be deleted at a later time as mentioned, so that a skeptical user looking for bogus policy files will not find it.

Exploiting the Bytecode Verifier: The Java Native Interface

It could also be feasible, with a compromised bytecode verifier, to generate bogus bytecode which performed operations similar to those described in Thompson's article.

Possible Exploits

The Java Native Interface allows Java code to call native methods (i.e., libraries written in native code) from a Java applet or application. Ordinarily, running a native method (or loading a native library) is forbidden by the sandbox. However, there are ways around this, one of them being the `enablePrivilege` method, which results in the user being prompted to allow an applet to run out of the sandbox. If the code is properly signed, or if the user allows the applet to run outside the Java sandbox in Netscape, the applet can load native libraries without a problem, thus enabling an applet to execute unverified code (i.e., which could be written in C, C++, or several other languages, including assembly).

This possibility leaves the door wide open to attacks, similar to those described in Thompson's paper. Other attacks along the lines of a native code Trojan horse are conceivable; the key to such an exploit would be figuring out a way to trick Netscape (or the appropriate appletviewer) into loading the appropriate native library, where the Trojan horse could be placed. Using such a native code implementation, an attacker can bypass the bytecode verifier and is free to implement whatever he chooses.

The inconsistency which makes this hack possible is that native libraries which are loaded and run by the Java VM are not verifiable. As such, it is possible to maliciously alter native libraries which are called, inserting the Trojan horse into non-Java code run by the Java VM. Running exploits through native code have other implications, including those involving authentication and authorization as discussed in the section on JAAS below.

Java Authentication and Authorization Service

There is currently an early access release of the Java Authentication and Authorization Service (JAAS), which is designed to enforce access controls on Java programs based on **who** is running the code (in addition to access controls which already exist based on code signatures and codebases). This brand new API provides support for authentication and also extends the existing authorization components in order to enable it to enforce access controls.

These classes (especially the provided login modules) are dependent on native libraries. For example, the `com.sun.security.auth.module.SolarisSystem` class loads a native library to access underlying system information, such as `uid` and `gid` for a particular user.

This could be hacked such that the underlying native library which the class believes it is loading is actually a bogus version of the class. For example, I could create a library which simply returned `uid=0` (i.e., root) for my username and trick the module into calling the method in this library instead (conceivably by simply giving the bogus library a valid name – this, however, could have catastrophic implications for the underlying operating system as well, however...). A similar trick is probably also possible with Windows NT and the `NTSystem` class.

Covering Tracks

In the case of loading native libraries, it's quite feasible for a user to load a native library, such as the libraries required for JAAS, and have the VM cover its tracks. In fact, assuming that the library could be loaded at all, it could likely cover it's tracks in a similar manner as Thompson's program. In the case of loading an intentionally buggy library, it is likely that the class loader will not be able to verify the validity of a native library, especially if this verification is being performed completely in a Java environment. Additionally, since the native library will likely be written in C, the "buggy" source code to the library can be covered in precisely the same manner that Thomson hid the bugs in his source code.

Exploiting the VM

There exists a possibility that an attacker could cause the user to think that a particular VM is running, when another totally bogus VM with any of the necessary components (class loader, bytecode verifier, etc.) compromised is actually running. This, of course, would allow the attacker to bypass all of the security features which Java provides. This comes down to an important question: can you really trust that the VM on which you run an applet it actually a legitimate VM?

Possible Exploits

The specific method by which Thompson describes the bugs being reinserted into the code as it is compiled and subsequently deleted following compilation could be accomplished simply in similar ways in Java; for one, one might alter the VM to insert instructions according to the attacker's plans, independent of the original Java code and after the bytecode has been verified. This issue comes down to whether or not we wish to trust the VM, and how trust of the VM is managed.

A comparable attack might be a VM which alters the bytecode after verification but subsequently leaves the bytecode as it originally was when it was loaded and verified. Given that the VM itself can be compromised in so many ways, this is only one possible example of a VM-related exploit.

Covering Tracks

If the VM itself is written in Java, there are ways to prevent disassembly, including a program known as "Crema" which scrambles symbolic information in class files making its own references between classes and library packages so that the JVM can still read the class file, but it can (theoretically) no longer be decompiled. This will, at the very least, prevent bugged source from being decompiled by a skeptical user. Currently, however, most VM's are written in C/C++, thus making the trick described in Thompson's paper applicable in this case.

Problem 2-3. Queso con Crackers? (Raquel Hill, Adon Hwang)

The security implications of TCP-fingerprinting programs depends on the user base of those operating systems. In general, for a technically-capable user, fingerprinting in of itself does not provide a great threat, assuming that the security holes in the operating system and applications are taken care of. However, for the general public and large organizations whose expertise may not match those of us security experts, fingerprinting may give the attacker an upper hand. Fyodor, author of NMap, describes certain application vulnerabilities that the attacker can exploit only once since "a failed attempt will crash the [bind] daemon"[1].

Let us elaborate. The ability for an outsider to fingerprint a system is not necessarily a vulnerability nor a threat. It is what the attacker does with that knowledge. Presumably, given the identity of the system, the attacker will proceed with attacking the known vulnerabilities for that particular system. Normally, an attacker will need to try a systematic trial-error process in attempting to infiltrate the system- i.e. different operating systems have different bugs. This may be an arduous process. The fingerprint information will shorten this period of blind system probing.

Now, if the administrator of that system has already taken caution to close those bugs and taken some measures to secure the system in anticipation of attacks (which we elaborate below), the attacker may not succeed in infiltration. For example, a secure firewall or filter may deny unauthorized users access and filter out these malformed packets that are used for fingerprint probes. A traffic shaping device may alter the timing of packets such that precise identification is difficult: the filter may be a passive device without any connection to the network itself, immune to attacks to itself; the device may alter packets through two pass-through ports, much akin to a router. Administrators may alter the network stack itself (given the source code to the OS) or may replace the stack (if possible), to alter the fingerprint. The network may be actively scanned for unusual packets- Christmas-tree packets, unauthorized NFS packets or NIS requests, SYN portscan probes, etc.- using tools such as inflog (Infoworld 10.26.98 [could not find link at rootshell]). Note that legitimate packets such as normal SYNs cannot be blocked unless an active filter mediates. However, knowledgable administrators may be rare- not all systems used by the general public nor large organizations may have these precautions in place.

At any rate, the most sure countermeasure is changing the behavior of the operating system in question at the kernel level. It is difficult to safeguard all systems against all possible fingerprints without specifically altering the network stack itself. Malformed packets may be filtered out, but methods such as TCP initial sequence number sampling are more difficult to guard against without changing the stack.

We should note, however, that denial of service attacks are not necessarily immune to these countermeasures on fingerprinting. A public web server cannot distinguish between legitimate TCP connection request SYN's and malicious SYN floods. ICMP ping floods cannot be filtered without disabling ICMP echo packets on the network itself.

Countermeasures against these fingerprints are not necessarily just "security by obscurity." Countermeasures such as active scanning or pattern-detecting firewalls may spot acts of intrusion while the attacker is probing for vulnerabilities. Without such safeguards in place, the attacker is free to take time and probe all known bugs. It is difficult to have a system that is bug-free. Countermeasures to fingerprinting cannot hurt. Especially when the general public or large organizations are concerned, packaged countermeasure tools, analogous to the downloadable fingerprinting tools, may allow a larger portion of hosts on the Internet to be guarded.

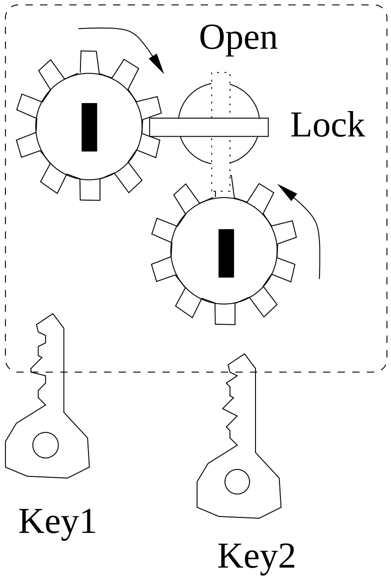
[1] `http://www.insecure.org/nmap/nmap-fingerprinting-article.html`

Problem 2-4. Physical Analogs of Crypto Primitives

Physical public-key crypto (Chiuyi Chan, Ying Lee, Ding Meng, Tairan Wang)

Two-key-lock

It would be quite easy to design a mechanical lock that have two keys. One is used to lock it, and the other is used to unlock it. In this case, the two keys are truly uncorrelated and knowing(having) one key does not allow one to figure out the other key. In this respect, it is quite different from the public-key crypto, where the two keys are related and can be broken through brute force.
However, the physical nature of this lock also severely limits its usefulness. One great feature of the electronic public-key crypto is that the public key can be duplicated and distributed easily, allowing many people to use it. The mechanical key to our lock, on the other hand, will be quite cumbersome to duplicate and distribute. The fact that one has to be physically at the lock also reduces the amount of people that can have access to it at one time. It would perhaps be an advantage, if limited access rate is desired. In the case where only one public key is available, then we will have a "token" system, where only the one with the public key is allowed to open the lock at one time.



Functional analogy and the mail box

The public-key crypto is used to secure communication between people, so that messages intended for the receiver can not be read by a third party, while anyone with the public key of the receiver can send the message. A functional analogy in the physical world would be a device to allow anyone to send physical things to someone without a third party knowing what that something is.

There is actually an existing system that follows this closely. It's the mailbox! (those kinds with a slit and a lock) Anyone can drop a letter into the box, but only the owner can open the box and retrieve the letters. The public key in this case is the narrowness of the slit, if you will. One can also easily imagine using a two-key-lock in this case where the public key "locks" the box in the sense of pushing the content into a receiving area accessible only by the owner. If we consider on a grander scale, the whole mail delivery service is like a public-key system for everyone, with the exception of noisy mailman and neighbors.

Of course, the reciprocal nature of the public-key/private-key pairing is lost. We won't be easily encrypt a package containing another encrypted message either, unless we have small portable mailboxes. This example also deviates a bit from the "mechanical lock" picture we started with. We feel it worth mentioning because of the close functional analogy it provides.

Physical Hash Function (Andy Berkheimer, Raffi Krikorian, Luke Matkins)

Such a function would take an object in as an input and return a concise physical identifier for the object.

Possible implementations: Such a function is ambiguous due to difficulty in defining what exactly an object is. For example, if you take a glass of water and replace one hydrogen atom with another, is the new glass of water the same as the old? We'll assume for this exercise that they are indeed the same object, but this is certainly a debatable point.

We could not think of a perfect solution to this problem, but following is a list of some of the ideas we considered.

- last digits of object's weight: an object could be weighed with a very accurate scale, and the last n digits of the weight could be printed and used as the hash value. If the scale is accurate to a minute amount, it would be very difficult to create a false object with the same weight. An attacker using some sort of nanotechnological device might be able to fool this system, however.

- object's response to low levels of radiation: an object could be bombarded with (hopefully) non-intrusive levels of radiation, and its echos could be recorded and used as a hash function. This approach may be destructive, however, and would probably be dependent on the position of the object when being exposed to the radiation.
- other destructive approaches: we considered a number of hashing functions which had the side effect of destroying the original object. These included tearing the item apart atom by atom. We decided to reject this class of functions.

Possible applications: Like its electronic counterpart, such a function could be useful in detecting message modification. Two parties wishing to exchange an object could exchange the object's hash value over some secure channel, and then send the object over an insecure channel, confident of being able to detect if the object was modified in transit.

If such a function were to always return the same value on an individual person, it could be used as a form of biometric authentication. Alternatively, it would make it possible to authenticate users based on extremely inexpensive keys (think lumps of coal).

Other implications: Such a function would be generally useful to anyone who wished to know if a new object is the same as any previously seen object.

Perfect physical security (Dilong Zeng, Adam Holt, Svetlana Shnitser, Igor Tarashchanskiy)

Implementing an unbreakable safe is simply not possible, unless contemporary material science changes very drastically! However, a very near equivalent can actually be built that quite reliably self-destructs on break-in –which accomplishes much the same result as the unbreakable safe (in either case the contents are unrecoverable, whether stolen in an unbreakable safe or destroyed by the vault.)

Novel application: disposal. In particular, the very real and very expensive problem of destroying Russia's vast anthrax reserves can now be cheaply solved! How? Bio and chemical weapons can be inserted in such (eternally!) unbreakable safes. For absolute disposal, similarly eradicate the key: place all existing keys to safe inside with biochem weapons (assumes self-locking doors, which indeed all safes seem to have.) Of course illegal disposal (and murder) become just as easy: the thief of an unbreakable box can place it (and any victims) in a larger unbreakable box, throw the keys in to jealously and provably deny you access to your crown jewels –and loved ones– forever!

Again, the consequences of a cheap unbreakable safe (for traditional safekeeping) are less than one might initially imagine if that safe can be unbolted from the floor and stolen. Given both denial-of-service attacks (safe may be stolen and held ransom, or be stolen forever out of spite.) Worse, you are still vulnerable to being coerced/tortured into giving up the key, so our perfect safe is looking less and less perfect. Indeed, until immovability and key-uncoercability are perfected, most people will continue to prefer to rent a bank's safety deposit box. Still, unbreakable safes are a step up from money in your mattress. Note: judicial consequences tend to run directly opposite of the above physical hashes –destruction of evidence suddenly becomes easy!