CS 261: Security: Lecture notes

University of California, Berkeley

Derrick Coetzee

November 23, 2009

Today's lecture addressed the problem of dealing with security in situations in which the attacker has physical control over part of a distributed software platform. A few examples of this phenomenon include:

- Gaming: One of the oldest examples. Here, a malicious gamer seeking to cheat has full control over their gaming client machine, allowing them to access any state visible to the client as well as issue commands on behalf of the client.
- AJAX web apps: These web apps use a combination of server-side processing and client-side scripting. A malicious user with a hacked browser can access any state available to the client-side script and issue any command on its behalf. A common error is to check input validity on the client side only.
- Tachographs: Tachographs are devices used to record the time spent driving by commercial truck drivers, used to enforce local regulations and prevent accidents. Companies looking for an economic advantage subvert these devices and mislead inspectors.
- Copy protection and DRM: The software designed to protect the content from copying resides on a device under the physical control of the person seeking to copy it.

1 Cheating in MUDs

A number of simple examples of cheating in games can be illustrated with MUDs, simple multiplayer online text-based role-playing games dating back to the 1980s, and in particular the DikuMUD "hack 'n slash" servers. In these games, each player would enter textual commands such as "go north" to move between locations and see textual descriptions of those locations, as well as interact with any other players in the same location. They could manipulate items, attack monsters, and develop their characters by gaining levels and skills. By forming groups with other players, they could combine their skills to take on more powerful enemies.

In some ways, the DikuMUD platform was close to ideal in its security model: since the client operated directly over a standard TCP connection and acted only as a "dumb terminal", the client has no access to any information that the player does not, and has no privilege that the player does not; all input/output is mediated by the server directly.

On the other hand, precisely because their interface is so simple, it's straightforward to write sophisticated MUD clients on top of that standard TCP connection that can issue commands on behalf of the player, what Matthew Prichard calls "reflex augmentation" in his "How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It." Client mechanisms included *triggers*, which match regular expressions to the received text and produce commands

instantly in response, and timers, which allow players to consistently predict when the server is about to increase their health and go to sleep, allowing them to gain a greater amount of health. As in many game contexts, certain uses of these augmentation tools was banned as cheating, whereas other uses were considered essential for expert play. Additionally, augmentation could be exploited by other players who had no access to sophisticated clients by intentionally triggering the triggers used by other players, causing them to unwittingly take action on behalf of the exploiting player.

Another problem with MUDs was the number of simple bugs and design loopholes that could be exploited for an individual player's gain. A common example is that if a player wanted an item, but that item had already been taken, they would exploit a bug to crash the server and cause it to reappear. The DikuMUD server in particular, written in C, was rife with easy ways to crash the server, including null pointer bugs, memory leaks, integer overflow, buffer overflows, and infinite loops, including one unfortunate oversight involving attempts to put a container such as a bag inside itself.

2 Cheating in other games and defenses

When complex graphical interfaces come into play, it's no longer possible to maintain the MUD's ideal of the "dumb terminal" - because of performance concerns, the need to update the display rapidly without an expensive network communication with the server, the client is forced to cache state that it may need in the near future to show a consistent view to the player. If the client possesses this data, then this data can be extracted from the client by a skilled cheater and made visible to the player prematurely. This confidentiality problem is endemic in both first person shooters, in which "wall hacks" enable players to see other players who should be obscured by obstacles, and real time strategy games, in which the client typically possesses the state of the entire game world and revealing it provides a major tactical advantage to the player accessing it.

Another problem common with complex game clients is what Prichard calls *authoritative clients*, in which the security of the game relies on the clients restricting in some manner the messages they send to the server. A compromised client can send any message it chooses, and if the server trusts this information (for example if it says "I have just won the game") then the player can achieve elevation of privilege. This is particularly problematic in fast-paced games such as first-person shooters, where events such as one player shooting another combine distributed state and there is insufficient time to reach a distributed consensus; a client is forced to reach a decision before the server can report a result, and it's unclear what should happen if these results are inconsistent.

As with any security system where the attacker has physical access, there is virtually no way to guarantee security *a priori*, particularly on a general-purpose PC platform. Penalties such as banning can be effective deterrents, particularly if they include some form of ID verification. Cheaters can be detected fairly reliably using statistical techniques that estimate the likelihood that a player is cheating, at the expense of including some false positives (typically, highly skilled players). It may also be helpful to restructure the goals of the game so that cheating is not rewarded.

On gaming consoles, which are dedicated devices built to run a single game at a time, attestation frameworks such as "Trusted Computing" can help to make devices tamper-resistant by ensuring at each stage of the booting process that the next component to be loaded has been approved for use on that device. Additionally the device relies on encryption of all transmitted data with a tamper-resistant cryptographic module. Although a deterrent, such schemes remain vulnerable to vulnerabilities in game software, typically through manipulation of complex saved data, as well as physical tampering. Additionally, there is no authentication or encryption for

input devices, due to the market for third-party add-ons, making reflex augmentation attacks relatively straightforward.

A simple commonly-used tactic is to periodically scan the state of the system, such as running processes, and send the results to the server. This allows the game operators to react to widely-deployed cheating software by automatically banning players who are running that software. Since cheating processes can attempt to hide themselves from other processes with rootkits, or to modify or disable cheat detection software, cheat detection software itself sometimes employs rootkits to attempt to bypass these measures.

3 A difficult-to-circumvent scheme for hash validation

Another simple deterrent is for the program to periodically take a hash of the complete state of the client, or a statistically random portion thereof, and transmit them to the rest of the system. This remains vulnerable to changes to the client-side code associated with producing and distributing the hash. A scheme has been designed which attempts to make this more difficult; it operates as follows. (citation for this method?)

Consider a hash object with some internal state (an integer) and two methods, get() and put(x):

- The get() function hashes the current internal state, stores it into the internal state, and then returns the internal state, acting essentially as a pseudorandom number generator.
- The put(x) function hashes the current internal state together with a new piece of state x and stores the result back into the internal state.

Now we can define our statistical hash validation function:

1. Receive a nonce from the server and put(nonce).
2. Do the following a large (e.g. 2^{36}) number of times:
   (a) addr ← get() *// get a pseudorandom address*
   (b) b ← *addr *// Read the byte at that address*
   (c) put(b) *// Incorporate that byte into the hash*
3. Do get() and send that value back to the server.

Moreover, it's an important requirement that the functions get() and put(x) are hand-tuned to run as fast as possible for the particular platform. If the attacker attempts to inject code to hide their modifications from the hash validation, this will slow down the computation enough that by the time all iterations are complete, the server will receive the response significantly later than expected, even taking network latency into account.

This scheme has not been deployed in practice - primarily because the task of writing a hash function implementation that is optimally fast for a particular machine is an a difficult and specialized task, and attempting to do so for many types of PCs is impractical. The scheme needs to be modified to cope with the straightforward attack of modifying the page tables during each hash validation to map out any modified pages and map back in the "good" ones. Additionally, because it depends critically on assumptions about how quickly the client can run the hash validator, it remains vulnerable to misreporting of system characteristics such as CPU speed and I/O bandwidth, which can be facilitated both by modifications and by hardware overclocking.

4 The arms-race model

In contexts where the attacker has physical access to a part of the secure platform, traditional strong theoretical guarantees of security may become infeasible or impossible to achieve. In these settings, the next best thing may be the *arms-race model*, a system in which attackers and defenders constantly react to one another; attackers invent new attacks, and defenders mitigate or complicate enough attacks that the total amount of circumvention is kept relatively low. If the defender has enough resources, they can make most attacks too expensive to be worthwhile. This is acceptable in low-security contexts like DRM or gaming where some limited amount of circumvention has acceptable impact on large-scale economics (overall enjoyment of the customer base in the case of cheating, or overall income from content purchases with DRM, for example).