

ON YOUR GUARD

CHAPTER OUTLINE

Malicious Use 20

- Turn Off Listeners When You Don't Need Them Anymore 20
- Set a Minimum Delay for Accepting Input 20
- Detect Malicious Use and Shut Down the Game 21

Data Protection 21

- Memory Hacking 21
 - Hash Data 22
 - Breakup Data 24
 - Insert Red Herrings 25
- Protecting Sent and Received Data 25
 - Hashing 26
 - Ciphering 26

SWF Protection 27

Summary 27

No matter where your game is hosted on the Internet, if people come and play it, someone will invariably attempt to hack it. Don't take it personally—some people are just jerks. Hacking can include everything from abuse (like jamming the game with input all at once in the hopes of crashing it), botting (where someone writes a program to play your game for them), and harvesting/changing data both in memory and information sent over the Internet. These jerk people are motivated by a variety of factors. Perhaps you are giving away some prize and they want to cheat to win it. Maybe there is a scoreboard associated with your game and they want to be number one. Sometimes, they just want to prove to their other hacker jerk friends that they could do it. The hacker community doesn't let you rest on your laurels; if you haven't hacked something *today*, you're not worth the smelly couch you sleep on in your mother's basement. Sorry, I digress.

There are a number of strategies you can use to make it very difficult for someone to hack your game. Note I did not say you could prevent it 100%. Let me stress this point: *no* game is unhackable. To make such a claim would be to invite a whole host of the best jerks

to prove you wrong. The goal with these strategies is that with every roadblock you introduce, a few less people will be willing to try. The techniques I will outline in this chapter will cover two main areas: malicious use (which includes abusive behavior and botting) and data protection.

Malicious Use

Some games are more susceptible to abuse by players than others. Games that rely heavily on keyboard input, such as many puzzle games, are particularly prone to being attacked. This is because it tends to be easier for hackers to write bots (programs to do work for them) and other scripts for keyboard-driven games. It's harder to track where the mouse is on a screen because that can vary from machine to machine depending on resolution, position of buttons, etc. Here are some tips to help block keyboard hacking:

Turn Off Listeners When You Don't Need Them Anymore

It's often very tempting when developing, particularly under a deadline, to just add listeners to things such as keyboard input at the onset of a game and then just remove them at the very end. Unfortunately, giving people uncontrolled input can open you up to tampering by bots and other scripts. If you're expecting input from a user, add a listener only when you're ready to receive it and turn it off after you've received your first piece of input. This allows you to analyze the data you've been given before opening the "pipe" again. This is another place where weakly referenced listeners are very important. When objects that weak listeners are attached to are tagged for garbage collection, the listeners will get removed automatically. Refer Chapter 4 to learn how to use weakly referenced listeners.

Set a Minimum Delay for Accepting Input

Often bots are used to speed up the progress of a game faster than a human would be able to accomplish it. If you define an interval (which depends heavily on the style of game in question) during which the listener ignores input, you can force hackers to slow down their bots to a human rate of speed. Here is what code for this might look like:

```
import flash.utils.getTimer;

private const MINIMUM_INPUT_DELAY:int = 150; //milliseconds
private var _timeOfLastInput;
```

```
private function onKeyDown(e:KeyboardEvent):void {
    if (getTimer() - _timeOfLastInput < MINIMUM_INPUT_DELAY)
        return;
    _timeOfLastInput = getTimer();
    //OTHER INPUT-RELATED CODE
}
```

Detect Malicious Use and Shut Down the Game

This is a more drastic measure and probably only worth implementing if (1) you have a legal obligation to other users to prevent hacking (because of prizes or money) or (2) you know that certain people are hacking your game in a particular, consistent way. The second of these two criteria is harder to define and can lead to nonoffending users reaping a consequence designed to catch hackers. There is a balance to be struck with this approach, particularly if you have less than adequate data to prove when someone is hacking your game. Occasional rapid input on a keyboard-driven game could just mean the player is genuinely skilled. You don't want innocent players to have a lousy experience just because you were trying to stop the jerks.

Data Protection

Although bots and malicious users are certainly a concern, they tend to be less of a problem than players who attempt to manipulate the data inside, being received, or being sent from the game. The next section will discuss ways to protect your game data, both in memory and “over the wire.”

Memory Hacking

Writing a bot for a specific game can be a time-consuming effort, and as I mentioned above, they cater more toward a specific type of game. It would be extremely difficult to bot a side-scrolling action game because the bot would have to know a lot more about the changing game screen than is probably possible. In these cases, a hacker is more likely to just try and manipulate a few values in their computer's memory in the hopes of giving themselves a high score. There are several utilities that exist which will allow people to modify memory addresses, so a decent hacker won't have any trouble finding them. It would be dangerous to just start changing memory values blindly, so the way these programs tend to work is by analyzing which memory addresses have changed over a period of time and telling the user what is in each one. If a hacker starts the utility and then plays a couple of turns in the game, they will probably be able to narrow down relatively quickly where data like their scores or progresses are kept. They then use the same utility

to change the value in their favor. At this point you may be saying, “Well, how am I going to combat *that*?” Don’t worry, there are a few things you can do which will make the jerks’ jobs a lot harder.

Hash Data

One way around memory hacks is to “hide” vital data you’re storing. You can do this any number of ways, but one effective method is to use a technique known as *hashing*. A hash is a form of cryptography intended to validate a piece of data by running a series of procedures on it. Depending on the hash algorithm you’re using, the result of a hash function will be a string of the same length every time. If you’re scratching your head right now, don’t fret; we’re about to look at some examples. For instance, a common hashing algorithm is known as MD5. When given data of any length, it will return a 32-character series of letters and numbers. Think of this as that piece of data’s *signature*—every time an MD5 hash is run on that data, it will produce the same result. As an example, the string “Real World Game Development with Flash” will always output:

```
37afb91b7a3c8042f5d0253f81cf7c5
```

My name will always result with:

```
e640754e479fa16c1320c97e65ccdb13
```

It’s important to note that these hash results are not guaranteed to be unique, that is, two different pieces of data could feasibly result in the same 32-character string. This phenomenon is known as *collision*, and some hash algorithms are more susceptible to it than others. MD5 has some weaknesses where this is concerned, but it is also very fast, so if you are changing values regularly over a period of time it is probably worth the decreased security. Another method, known as SHA-256, is known to have a lower collision probability, but it is slower to process. It is more suited to data that is being sent outside of Flash, which we will discuss shortly. Now let’s look at a practical example of how you could use a hash to protect a game’s score.

Consider the following very simple class. It stores a number in a private variable through public accessors.

```
package {  
  
    public class Game {  
  
        private var _score:Number;  
  
        public function set score(value:Number):void {  
            _score = value;  
        }  
    }  
}
```

```

        public function get score():Number {
            return _score;
        }
    }
}

```

This code contains no protection from hacking and the `_score` property is very susceptible to having its value changed. We'll now introduce an MD5 algorithm. You can find many different implementations of MD5 on the Web—most that are in ActionScript started as Javascript. I have included one in the Bonus Chapter 2 examples folder. You can use it along with the sample `Game.as` file.

```

package {

    public class Game {

        private var _score:Number;
        private var _scoreHash:String;
        private var _scoreDate:Date;

        public function set score(value:Number):void {
            _score = value;
            _scoreDate = new Date();
            _scoreHash = MD5.hash(_score +
                _scoreDate.toString());
        }

        public function get score():Number {
            if (_scoreHash != MD5.hash(_score +
                _scoreDate.toString())) return 0;
            return _score;
        }
    }
}

```

Now we're getting somewhere. Each time the score property is set, it will not only store the actual value, but a hash of the score to check against when reading the value back out. MD5 and similar hash algorithms work better when you have more data to encrypt as it makes it less likely a matching string that can be generated by something else. Since a score is likely to be only a handful of characters, we also generate a timestamp from a `Date` object (yet another way to try to make the sequence unique) and append it to the score. In this instance, the date is what is known as the *salt* of the hash. It is the piece of data that is irrelevant to what you're trying to protect but is used to further obfuscate it. This way, when the score is set, three different values have been written to memory.

If any one of them changes, the score will be invalid when it is read back out. In this case, if the hash doesn't match, I return 0 instead of the actual score value. You could even throw an error if you wanted to, instead of returning anything, but it's important to consider how you want the game to behave once illegal activity is detected. A big advantage of this method is it is highly customizable. There are any number of properties you could use as the salt, and they need not be the same per game or even the same per class. In fact, the more you vary your use of them, the harder it will be for a hacker to determine how you're generating the hash at any given moment. Another unique identifier you could use as a salt is the server string for user's computer. It is a property of the Capabilities class and produces a nice long string of many values for that machine.

```
import flash.system.Capabilities;
//
_scoreSystem = Capabilities.serverString;
_scoreHash = MD5.hash(_score + _scoreDate.toString() +
    _scoreSystem);
```

Once again, this does not eliminate the possibility of someone still managing to hack your game, but it does make it a bigger pain for a hacker than moving on to a game that is an easier target. You also need to weigh performance against how many values you want to hash. Usually a couple of main values is enough. Values that change very frequently (like every frame) will be harder for hackers to target anyway, and the added processing required to run MD5 multiple times a frame could get heavy on slower machines.

Breakup Data

Another way to help obfuscate data is to break it into pieces and reassemble it only when you need it. This method is particularly helpful if you have strings in your code that contain important information, like a key or passwords for levels. By breaking up the string into single characters in an Array, for instance, you help obscure your data from memory readers. Any single piece of the data is likely to be useless on its own and assembling it only when you need it (and subsequently discarding the assembled version) ensures the value can't simply be fished out of memory.

```
private const PASSWORD_LEVEL_1:Array =
    ["H","A","C","K","E","R","S","U","C","K"]
```

With the above example, a simple toString() call on the Array will return you the reassembled value, but in memory it will store 11 different values.

Insert Red Herrings

This method can go off the deep end if you're not careful, but another avenue to take is to insert meaningless data along with the data you care about. The idea here is to have so much data moving around and changing any given moment that a memory utility can't differentiate between the real values and the "noise." The catch is that unless you have a solid strategy for making this distinction yourself, you run the risk of falling victim to your own defenses. This is also quite a bit more hassle to implement, so it should be reserved only for information that is extremely critical. If we were to build on the previous example, we could store random numbers in among the real data and then store a separate Array of values that map to the correct indices in the first Array.

```
private const PASSWORD_LEVEL_2:Array = ["H", 1,"A", new Object(),
    "C","K", new Date(), "E","R","S", .5,"S","U", new Object(),
    "C","K"];
private const PASSWORD_LEVEL_2_MAP:Array = [0, 2, 4, 5, 7, 8, 9,
    11, 12, 14, 15];

public function getMappedValue(name:String):String {
    var str:String = "";
    var stringArray:Array = this[name];
    var mapArray:Array = this[name + "_MAP"];
    for (var i:int = 0; i < mapArray.length; i++) {
        str += stringArray[mapArray[i]];
    }
    return str;
}
//
var password:String = getMappedValue("PASSWORD_LEVEL_2");
```

As you can see, this is quite a bit of trouble to go to for very many values and also begins to eat a considerable amount of memory to store just a simple 11-character string. Ultimately, you must weigh the cost of protecting your data.

Protecting Sent and Received Data

More and more Flash games out today are making use of external data, both for loading content and for posting data (like leaderboards). I discussed some of these techniques in Chapter 10 using XML. This data is very vulnerable to tampering, even more than that in memory. Anyone with a basic type of HTTP activity monitor can see the data in plain sight coming in and going back out. The best method for protecting this data is encryption, and there are a few different approaches to doing so.

Hashing

Much like the previous memory examples, using a hash algorithm to validate data that is sent to a server from Flash is a great way to secure it. Depending on the nature of the data, you may want to use an even more robust hash, such as SHA-256. It is slower to process than MD5, but usually data transactions are things such as score postings or saving profile information: events that do not repeat rapidly in succession. When you use a hash to send data to a server, both the Flash and the server need to have two things:

- The salt for the hash.
- The order of operations to recreate the hash.

Here is an example of how you might send out a score to a server.

```
private const LEADERBOARD_URL:String = "http://www.flashgamebook.com/savescore.php";
private const LEADERBOARD_SALT:String = "hackerjerks";

public function saveScoreToServer():void {
    var urlVars:URLVariables = new URLVariables();
    urlVars.score = score;
    urlVars.date = new Date().toDatestring();
    urlVars.checksum = MD5.hash(score + date + LEADERBOARD_SALT);
    var urlRequest:URLRequest = new URLRequest(LEADERBOARD_URL);
    urlRequest.method = URLRequestMethod.POST;
    urlRequest.data = urlVars;
    var urlLoader:URLLoader = new URLLoader(urlRequest);
}
```

This method would send the score and timestamp, along with the hash to check against. The server would need to know that the salt is “hackerjerks” and that the three properties must be concatenated together and hashed to match. This works well for data that you don’t mind people seeing but want to validate as legitimate once it arrives at its destination. This isn’t a solution for loading data that you don’t want to be seen at all, such as solutions to a puzzle or sensitive user data being loaded from a database. For these instances, we turn to a method known as *ciphering*.

Ciphering

Unlike a hash, which is a one-way encryption (there is no way to get back to the original value from a hash), ciphering is a two-way encryption. It turns the data in question into different data and then back to its original state. Most ciphers (the worthwhile ones, anyway) have a key associated with them that both sides must have when working with the data. A particularly popular and

powerful encryption cipher is the Advanced Encryption Standard (AES). Ciphering is even slower than hashing and results in a string longer than the data initially inputted. Here is what the name of this book would look like encrypted in AES with my first name as the key.

```
/pv/JFamBEGYRkrA8J8aIpceB+IqUb6XC11UPR8v1SNe7bugwDNtMhqD4J0LWP8g
```

As you can see, it is totally unintelligible from the original data. Without the key, it would take a *long* time for even the best applications to crack the encryption. For games with external puzzle or other textual content, running the data through a cipher like this before putting it on a public server is a very good idea. Other ciphers are less heavy on the processor than AES. Algorithms such as Extended Tiny Encryption Algorithm (XTEA) or RC4 are suitable for encrypting non-critical data and are much faster. I have posted a link to a free encryption library for AS3 on www.flashgamebook.com. It includes all the popular hash and cipher methods you probably ever need to use.

SWF Protection

The last item we'll look at in this chapter is probably the one most out of our collective control, protecting the code inside your SWFs. A number of utilities exist to extract ActionScript from a SWF, making it plainly readable, albeit without comments. Adobe has yet to seriously tackle the issue of SWF security, and as a result just about every SWF is vulnerable, regardless of Flash version. A few companies have written software that obfuscates the data inside a SWF to make it harder to extract. Two such programs are SWF Encrypt by Amayeta Software and SWF Protector by DCom-Soft. While far from perfect, they do an admirable job of at least making hacker's jobs harder by turning the output of most extraction utilities to garbage. In the end, running a program like this and selecting "Protect From Import" in the Publish Settings of your SWF are the best options if you have sensitive data inside your game. Hopefully, Adobe will work to remedy this lack of security in future versions of Flash. Until then, you can at least lower your likelihood of being hacked. You have to remember that these jerks are usually lazy and will go for the low-hanging fruit. If someone else's work is less protected than yours, they are a more likely target than you.

Summary

Now you have a number of new methods in your arsenal for protecting your work. Hacking and security on the Internet is

a continual arms race. As one company finds a new way of securing data, another individual will find a way to expose it. The best you can do is to protect yourself as much as possible. Well, and if you ever meet someone who you discover is a hacker in their spare time, break their hands.