Walking on Water

A Cheating Case Study

Online games offer opportunities for malicious users with the necessary skills to turn a profit. Many game developers acknowledge and address these risks, but new games still use technologies whose security implications have yet to be publicly disclosed.

Aaron Portnoy and Ali Rizvi-Santiago
TippingPoint

The online gaming industry is booming. With millions of gamers connecting from countries across the globe, a lucrative market has emerged for both the companies developing the games and those seeking to subvert them. Games such as World of Warcraft boast subscription counts topping 10 million active users (www.mmogchart.com/Chart1.html), which represents a ripe target for potential attackers.

These unprecedented online communities have evolved into microcosms of the real world, developing economies in which even the in-game currencies have real-world value. Web sites have popped up allowing gamers to buy everything from high-level characters to information or programs that help them cheat and obtain advantages over other players. This translation from virtual worth to actual wealth has created opportunities for malicious users with the necessary skills to turn a profit. Although many game developers acknowledge and address these risks, new games are still emerging that use technologies whose security implications have yet to be publicly disclosed. In this article, we focus on Disney's Pirates of the Caribbean as a case study. We discuss the game's architecture and the risks associated with the choice of a dynamic language, and demonstrate the possible impact when someone exploits these weaknesses.

Complex Systems, Dynamic Languages

Massively multiplayer online role-playing games (MMORPGs) are often implemented as exceedingly complex distributed systems that run on a wide range of end-user operating systems and hardware. To alleviate some of the difficulties of supporting such a heterogeneous network of users, many game development houses have started programming their products in platform-independent languages. Traditionally, they developed their games in static compiled languages such as C or C++ to exploit runtime speed advantages. However, supporting multiple platforms can be a time-consuming task in the development cycle.

Today, we see games increasingly written in dynamic cross-platform languages, such as Python (www.python.org) and Ruby (www.ruby-lang.org). A list of popular releases already implemented in such languages includes Disney's Pirates of the Caribbean (http://apps.pirates.go.com/pirates/v3/welcome), EVE Online (www.eve-online.com), and Civilization 4 (www.2kgames.com/civ4/home.htm). However, the migration to dynamic languages carries with it unforeseen risks to intellectual property and, in many cases, makes it easier for malicious users to subvert the game.

Pirates of the Caribbean Example

Disney's game developers wrote Pirates of the Caribbean in Python, a dynamic, object-oriented programming language. Dynamic languages are defined as high-level languages that perform type checking at runtime, but many of them also support reflection, metaclasses, and runtime compilation. These features are of particular interest because they require a significant amount of type information to exist in the distrib-

uted application. This information can include variable and function names, class hierarchies and relationships, and even the source code's original file names.

In Disney's case, the code is largely distributed in one file, Phase1.pyd, which contains serialized Python code that, in turn, contains all the logic behind the client-side portion of the game. Developers often design games such that the client performs many of the intensive calculations dealing with physics and graphics. This design alleviates the need for a server to transport and then process large amounts of data over a network. However, offloading these tasks to the client crosses a trust boundary. If a client can modify local code that isn't later verified by the server, cheating becomes easy. Let's examine how a dynamic language's features can help someone efficiently discover the presence of such weaknesses.

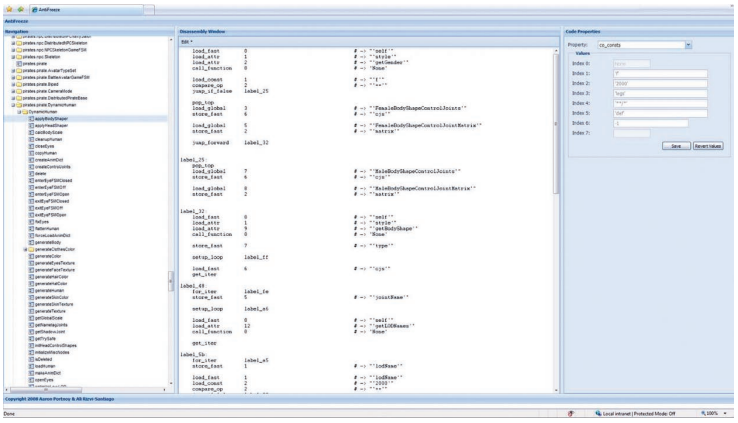

Figure 1. Code overview. A GUI-driven toolset such as AntiFreeze can help someone extract not only the program's logic but also type information, symbolic names, and even the class hierarchies and relationships as defined in a game's original source code.

Code Exploration

The process of exploring client-side code begins with demarshalling the serialized Python objects. We've released a GUI-driven toolset called AntiFreeze (http://code.google.com/p/antifreeze/) to make this process more observable. Upon loading a binary Python file such as Disney's Phase1.pyd, the AntiFreeze interface displays a high-level overview of the code and its structure, as Figure 1 shows.

For example, the figure shows that the source code defines a module called `pirates.pirate.DynamicHuman`, which contains a class called `DynamicHuman`. This class has methods `applyBodyShaper`, `applyHeadShaper`, `calcBodyScale`, and so on. Such information is available because developers wrote the game in a dynamic language, and the interpreter requires it—obtaining this information in a comparable game written in C that didn't include symbols would take a considerable investment of time and effort.

Code Modification

The middle pane in Figure 1 shows a function's low-level interpretation in byte code, which conveys the original source code's logic and is easily readable when disassembled. Here's an example:

```
load_const 179 # 'maxSpeed'
load_const 178 # 0.69999999999999996
load_name 195 # 'defaultMaxSpeed'
binary_multiply
rot_three
store_subscr
```

This code simply stores the value 0.7 into a variable called `defaultMaxSpeed`. Even in this straightforward example, we can readily predict the potential for abuse. Once we find interesting code—such as that which governs physics—changing it is trivial. In the previous code example, we can increase the value of 0.7, and thus a cheat is born. Reading through the disassembly of Disney's code in the Phase1.pyd file we find several interesting names:

```
pirates.ship.ShipGlobals
pirates.battle.CannonGlobals
pirates.battle.WeaponGlobals
pirates.economy.AwardMaker
pirates.economy.EconomyGlobals
pirates.piratesbase.PiratesGlobals
pirates.pvp.SiegeManager
pirates.quest.QuestConstants
pirates.reputation.ReputationGlobals
```

Many of these modules contain constant values that we can modify to affect weapon power, ship speed, experience points required to level up, and many more aspects of the game. The process for disassembling and re-injecting Python code into PYD files hasn't been documented extensively, which perhaps instills a sense of security in Disney's developers.

Malicious Opportunities

Figure 2 illustrates the result of code modification; here, a malicious gamer has significantly increased a character's jump height using the technique we just described.

Being able to alter an avatar's physics exposes other in-game side effects as well—for example, to save on calculations, the building structures in the game only detect collisions with their vertical walls, so if an avatar can jump high enough, it can fall through structure roofs, accessing otherwise inaccessible areas. Moreover, if a ship is at sea, a modified character can walk on wa-

(a)

(b)

Figure 2. Code modification results. (a) The avatar in Disney's Pirates of the Caribbean game is at the peak of a jump, clearly higher than surrounding buildings (the unaltered in-game jump height is closer to a meter). (b) A similar jump, but looking down on characters far below.

Figure 3. Cheating physics. After modifying Disney's Pirates of the Caribbean game code, malicious gamers can manipulate their avatars to walk on water after jumping overboard.

ter after jumping overboard (which is impossible without such a cheat). Figure 3 shows a modified character walking on water after having jumped off a frigate.

These cheats are so easily accomplished because the game is implemented in a dynamic language, and dynamic languages require type information to be present in order to function. Programs written in these languages run under an interpreter, so if the interpreter can access this type information, nothing stops a malicious user from doing the same. Programs written in compiled languages must be reverse engineered from assembly language, which can be significantly more difficult.

Another serious issue that arises with the choice of a dynamic language is its susceptibility to *botting*, which is the process of scripting a game such that it essentially plays itself, autonomously performing actions that increase the value of a player's character. These actions differ game to game but can include activities such as harvesting gold or defeating enemies to gain experience points. The ability to perform these actions without any human interaction allows a player to simply let the gameplay itself indefinitely.

Dynamic languages also support a feature called *dynamic recompilation* that allows for new code to be injected into the interpreter and evaluated during runtime. What this means for a game is that players can enumerate interesting functions defined in the developer's source code that control their avatars and call those functions themselves. A malicious gamer could write his or her own program to repeatedly access this functionality and quickly create valuable assets to sell to other players. The gamer could then expand this process to work over multiple accounts, essentially creating a profitable character farm.

As more developers make the switch to dynamic languages, the security implications we've described here will eventually need to be addressed. The availability of type information, dynamic recompilation, and introspection are the foundations of these languages, which developers should consider when implementing a large-scale application. Until mitigations come to fruition, the games in current release will continue to be susceptible. □

Aaron Portnoy *is a researcher in TippingPoint's security research group. His research interests include reverse engineering, vulnerability discovery, and tool development, and he's discovered critical vulnerabilities affecting a wide range of enterprise vendors, including Microsoft, Adobe, RSA, Citrix, Symantec, Hewlett-Packard, and IBM. Contact him at aportnoy@tippingpoint.com.*

Ali Rizvi-Santiago *is a researcher in TippingPoint's security research group, where his responsibilities include developing reverse-engineering-related tools and applying them to his daily tasks. Contact him at arizvisa@tippingpoint.com.*