Exploring Encryption in Networked Multiplayer Games Developed in Unity3D

Pat Healy[1]

Abstract— **This paper includes a casual exploration of a few encryption strategies for networked Unity games. The project included the implementation of a simple framework for hosting networked unity games through a Python-Flask server and encryption of traffic to and from that server with DES, Triple-DES, AES, and RSA. All but RSA were deemed feasible algorithms for regular multiplayer game data encryption.**

I. INTRODUCTION

Cryptography is not a topic that typically intersects much with the world of video game development, even in networked multiplayer games. Much of the internet traffic that drives networked games is either minimally encrypted or just completely un-encrypted. There are a few reasons for that.

- Game data isn't sensitive.
- Games often require low latency.
- Implementing encryption can take time, which developers often do not want to spend.

A. Sensitive Data

When determining cryptographic schema, we should consider both that the cost of breaking the ciphertext exceeds the value of the encrypted information and the time required to break the ciphertext exceeds the useful lifetime of the information. Of course, sometimes games require the player input sensitive information, such as when logging in or registering for a gaming service; these cases should very obviously be encrypted in a secure manner but this is not the topic I'm discussing. Think instead of the basic moment-to-moment gameplay of an online multiplayer game.

In these scenarios, the data transmitted simply contains information about actions taken by the given player. For example, it could contain the locations of any game objects modified by the player and some kind of identification number to identify the player and tie the action to them. Or, in the other direction, the server sends the client information about other game objects that move outside of the player's direct actions.

This is not data of particularly high value, at least in most cases. An attacker can only use it (specifically, they may use any player id data that authenticates the user) to impact the outcome of the particular game the user is playing, either by impersonating the player in packets sent to the server or impersonating the server in packets sent back to the player. This may negatively impact the player's performance in a multiplayer game, but this is ultimately very low stakes, outside of professional competitive play. Still, a large scale set of attacks could negatively impact players' willingness to purchase and/or play the game.

The useful lifetime of the data is approximately the length of a given multiplayer game session. A single session of most online multiplayer games lasts less than an hour, easily. Just to be safe, I'd say the time to break the ciphertext should exceed five hours.

B. Latency

Here we see a divergence between turn-based and real-time multiplayer games. Latency is effectively not a problem with turn-based games; therefore, the run-time of encryption and decryption is not a concern. In real-time games, latency is a concern; players expect game objects to be updated at a rate of at least 30 frames per second, meaning latency should absolutely not exceed 0.033 seconds. This is a complex issue, complicated by interpolation done on the part of the game developer outside of the exchange of packets (meaning transmission of 30 packets per

*A project for INFSCI 2170 with Dr. Prashant Krishnamurthy.

[1]PhD Student in Information Science, University of Pittsburgh

second is not necessarily required), so instead of a quantitative measure of speed, it may make more sense to focus on whether players themselves can detect further latency.

C. Implementation Concerns

The labor required to implement encryption in multiplayer games is clearly not trivial. Given the somewhat standard practice of not encrypting traffic at all, especially in the independently published video games made in Unity3D, implementation of a feasible cryptography schema will have to be incredibly easy and fast. Of course, this means taking advantage of existing encryption packages in the .NET framework.

D. The Plan

The goal of finding a feasible cryptographic schema for a multiplayer online game made in Unity3D is not to find the *best* algorithm but instead the easiest to implement algorithm that meets these requirements: (1) the addition of encryption should not introduce any human-noticeable latency, and (2) time to break ciphertext by brute force should exceed 5 hours.

This project is both an implementation of encrypted network traffic in a Unity3D multiplayer game and an assessment of several encryption algorithms within that developed framework.

II. Implementation

In this section, I've offered an overview of the network framework I built with Flask and Unity, the game I made to demonstrate it, and the encryption I included in the process. You can view the source code I discuss here at its public GitHub repository at https://github.com/PatHealy/UnityEncryptedNetwork

A. The Framework for Multiplayer Games

The multiplayer framework I created consists of a Flask server communicating with clients (Unity game applications) through HTTP. It is far from the "correct" way to do this kind of real-time networked game communication in Unity that is supposed to transmit at 30 packets per second, but it proved to at least be sufficient, and was well-suited for testing encryption algorithms in this context.

When the client launches, it first sends a GET request to receive the RSA public key of the server. Next, the client assembles an object containing the client's RSA public key, a string representing the client's chosen form of encryption for the bulk of network traffic, and a token representing the player's identity (i.e. "1" for Player 1); this object is then encrypted with the server public key and sent to the server. The server decrypts this data so that it now has the client public key, tied to a player identity token. The server then randomly generates keys for the client's chosen form of encryption, stores these keys and client public key in a dictionary that can be referenced based on the player identity token, and then encrypts the keys with the client public key so that they can send them back to the client.

Notably, this RSA encryption has a clear difficulty of plaintext size. RSA cannot encrypt a string larger in length than the modulus, and this packet from the client is definitely much larger than the modulus. I solved this by writing my own methods for breaking up the plaintext into smaller chunks and then reassembling these chunks post-decryption.

At this point, the key exchange has been completed in a secure way, taking advantage of public key encryption through RSA. Now, at every frame, the client will package up location data for all of the game objects this client controls, encrypt this data with the chosen encryption algorithm, send it to the server where it will be decrypted, the server will update a central roster of game object locations and then encrypt that roster to be sent back to the client, so that the client can update locations of other game objects controlled by other clients.

B. The Example Game

The example game is a simple two-player top-down racing game. The game is a simple race track with two cars. One client controls the pink car (Alice) and the other client controls the yellow car (Bob). So, to flesh out some specifics for the framework previously detailed, The Alice client will always send the location of the pink car to the server and receive the location of the yellow car from the server.

For the purposes of this demonstration, neither game client accepts significant player input. Both clients run their cars with a simple path-following AI. This is to ensure easy comparison between algorithms without being distracted by having to play the game.

Fig. 1. A screenshot of the Bob client running without encryption.

C. The Encryption Packages

Game traffic in this demonstration can be encrypted with five different algorithms: RSA, DES, Triple-DES, AES, or no encryption. The choice between these algorithms is decided right when the client application is launched. The two clients can choose to encrypt with different algorithms, which can make for simple comparisons.

One potentially interesting choice I made while implementing cryptography came in my choice to use ECB-mode instead of, for example, CBC-mode for DES, Triple-DES, and AES. This may seem like a poor choice for a few reasons, but I will explain my choice in later sections.

Encryption on the client side is done using the .NET framework's built-in Cryptography namespace. Encryption on the server side is done using PyCrypto.

D. Troubles with Implementation

Implementation took significantly more time than expected. This mostly came down to the difficulties in ensuring compatibility between data encrypted in different languages. There is shockingly little documentation in the realm of ensuring something encrypted with .NET AES in C can be easily decrypted with PyCrypto AES in Python. I spent many hours struggling over the specifics of encodings and the difference between byte-strings and arrays of bytes, as well as the difficulties of serializing any of these things for easy transmission over HTTP.

Of course, some limitations in time to implement (both because of my time wasted dealing with the previously mentioned issues and my own personal troubles being productive in the middle of a global crisis) meant that I was forced to cut features I wanted to implement. This lack of time is one of the reasons I ended up choosing to use ECB-mode for the block-ciphers (though definitely not the only reason). I would have loved to try more encryption algorithms than this short core list of four, and especially would have liked to try

III. Assessment

Originally, the assessment portion of this paper was planned to be much larger, but unfortunately some plans had to be changed due to the COVID-19 crisis. It was no longer feasible to run a small scale HCI study in which a bunch of human subjects try out the game with different encryption algorithms, because I no longer had access to the network of friends and colleagues I typically see in person on campus. Still, in my assessment I look to address each factor I found to be important in my introduction.

A. Ease of Implementation

As previously mentioned, implementation was more difficult than expected. It was still feasible a single developer to complete in under a week of full-time development, but it seems that a much higher fidelity solution still would be required for a real production environment. Considering the time crunch required of most independent games that use Unity, it seems this kind of encryption will only be feasible for game developers when it can be easily embedded in a higher-fidelity network framework in a more drag-and-drop way, likely requiring the development of a Unity Package. This may seem like a negative, but in fact it may be a business opportunity for a Unity asset.

B. Security Vulnerabilities

Like all systems, this one has its fair share of security vulnerabilities. I ultimately find them unremarkable, though.

The first of these comes as the server and client are exchanging keys. Let's say we have a client named Alice and our server is named Carol. We begin this process with Alice getting the public key from Carol and then encrypting Alice's own public key with it to send to Carol. In this there are two opportunities for a malicious actor, Trudy, to cause trouble. Trudy may intercept the initial transmission of Carol's public key and replace it with Trudy's own public key. Now Alice will encrypt her information with Trudy's key and try to send it to Carol. If Trudy can once again intercept this message, then Trudy could access whatever information Alice was trying to send to Carol. This would be a somewhat over-the-top man-in-the-middle attack, given the low stakes of this application, but it is technically possible.

Another malicious client, Mallory, could also just simply impersonate Alice from the beginning. Clients are only identified by the player identifying token that they submit when giving their RSA public keys to the server, so if Alice was truly "Player 1" but Mallory sent a message to the server first, claiming to be Player 1, then Mallory would be able to take the actions of Player 1 but not Alice. This is actually much less of a problem than it looks like. If Mallory is able to do this, she isn't really intercepting Alice's game, she is only playing the game instead of Alice, which does not technically enter the realm of cheating, our primary concern.

Beyond this, our main worry is that either a malicious actor could impersonate a player sending game object data to the server or impersonate the server sending data to a player. Without any encryption, the first of these is absolutely trivial (not even requiring and kind of interception), while the second is at least easy. Given the safe transfer of keys established with RSA, these tasks become much more difficult with any of the listed encryption methods (DES, Triple-DES, AES, or RSA). For the most part, these would require a brute-force attack, which necessarily exceeds the window of when this information is useful (i.e. the relatively short length of a match of the game).

In choosing ECB mode, as mentioned previously, I have exposed a possible vulnerability. Methods like CBC are typically used to protect against frequency analysis attacks. However, the data being encrypted is lists of floating point values (which represent locations), so a character frequency analysis would likely not be particularly helpful. Even still, any potential security vulnerabilities out of this would only decrease the time necessary to break the cipher, which likely would still take longer than the relatively short length of a multiplayer game. It is unlikely that the benefits of CDC in security would outweigh the implementation benefits of ECB in this context.

Ultimately, I find all four encryption methods perfectly acceptable to protecting against security vulnerabilities. The only wrong answer in this respect is to not encrypt at all, though obviously one could also rank the relative security of each algorithm in increasing security as No Encryption, DES, Triple-DES, AES, and RSA.

C. Performance

In this section, I focus on quantitatively measured latency across the five encryption methods, as opposed to qualitative analysis that I explore in the next section. Built into my implementation of the game clients, the client will automatically calculate a measure of average latency and display it on the screen. This latency measure is based on the amount of time it takes Unity to serialize data for transmission to the server, encrypt it, send it to the server, have the server decrypt it, update the stored game data, serialize this game data to transmit back to the client, encrypt it, send it to the client, have the client decrypt it, and update game objects. Basically, this is the time it takes for the client to complete a full update of game data, corresponding with the server.

I've measured a set of average latency values, based on running the client on my own windows machine while nothing else was running on the machine. Of course, there is likely some significant variance in these values, as plenty of things can affect the run-time of these algorithms that are beyond my control. This is a measure of wall clock time when really what we want is system or user time, but still they may offer insight in comparison.

This chart largely suggests that most of these encryption algorithms make no significant impact on latency. Considering their apparent impact on performance is one of the reservations to have

Algorithm	Avg. Latency (seconds)
No Encryption	0.0251
DES	0.0250
Triple-DES	0.0252
AES	0.0251
RSA	0.2210

about encrypting game data, this is a very good sign. The one very clear outlier of this is RSA, which has absolutely terrible performance, This latency is far above our stated maximum acceptable threshold of 0.033 seconds. All but RSA have acceptable latency.

D. User Testing

As previously noted, this was a portion of the project originally planned to be much larger than it ended up. Instead of a small-scale informal HCI study with a handful of Pitt students surveyed to see what differences they see in latency between the various algorithms, I was only able to perform a similar kind of survey with my two roommates.

I presented my roommates with a series of pairs of clients, each with one of the clients running without encryption and the other running with some encryption. I simply asked the subjects whether they thought they could notice a difference in visible latency between the two. Neither subject noticed any difference between unencrypted and both DES and Triple-DES encrypted clients. One subject claimed to notice a slight difference between unencrypted and AES-encrypted clients. Both clients noticed the incredibly poor latency in the RSA-encrypted client, which was significantly visibly worse than unencrypted.

Given AES's fantastic performance, as discussed in the previous subsection, I am somewhat skeptical of the one subject's negativity towards AES. Of course, with more human subjects this would be easier to understand. For all intensive purposes, I tentatively conclude that all of the listed algorithms except RSA are feasible in terms of avoiding human-noticeable latency.

IV. Future Work

A. Expanding this Experiment

If I was to continue this project, there are two main things I would focus on. First, I would expand both the breadth and depth of the cryptographic strategies I implemented. The algorithms I chose to use are simply very typical, which I think does not match the atypical nature of online game networked traffic, as I outlined in the introduction. It seems totally possible that a much simpler algorithm, which would still meet a security standard acceptable for this context, could run much faster and be much easier to implement. I feel my methods thus far are lacking creativity.

Secondly, I would have liked to expand the assessment of algorithms into the kind of rigorous Human-centric user study that I wanted to conduct for this project originally. It seems that a human-centric approach to this problem is necessary, since focusing on simple latency numbers obviously cannot tell the full story while games can effectively distract the user with interpolation heuristics.

B. The Future of Unity Online Multiplayer Games

This paper notably finds itself at a weird time in the world of online multiplayer games implemented in Unity3D. The UNet services, which historically have been the de facto way of implementing multiplayer games in Unity, have been deprecated but their replacement systems are still only in early development. This awkward situation is why I chose to simply develop my own simple multiplayer framework that fit the purposes of this project. My network framework with Flask should not be confused with a good network solution, but it was definitely good enough to serve this project, which has at least proven that implementing encryption into a networked game is totally feasible in .NET and perhaps the developers of this new official multiplayer framework should at least consider the pros and cons.

V. Conclusion

This work points towards the feasibility of a Unity3D package, or add-on to the upcoming new networked multiplayer framework, that embeds encryption in its communication. Though game developers often don't find it worth the work and performance costs to implement encryption, I've found that the security benefits are strong enough, and performance costs low enough, to make it a feasible option for developers, assuming they have the labor resources to implement it. Ultimately, if

this is to become a feasible feature, as I believe it should, it needs to exist either as a Unity-provided asset or one purchasable on the Unity Asset Store.