C

Cyber Security Protection for Online Gaming Applications

Wenbing Zhao
Cleveland State University, USA

INTRODUCTION

By online gaming applications, we mean both distributed applications that enable large number of users to play multiplayer games and those that enable people to gamble online because both types of applications could have huge financial stakes and the security and dependability challenges for both are rather similar. On the one hand, such systems must ensure continuous high availability so that users around the globe could play the games 24 by 7. This requires that the game servers be replicated to provide non-stop services. On the other hand, state-machine replication requires that the replicas be deterministic or rendered deterministic. This requirement does not work well with gaming applications because random numbers are essential to their operation. For example, in a card game, the random numbers are used to shuffle the cards. If the random numbers used are not robust, the hands in the card game may become predictable, which could damage the integrity of the game and may lead to financial losses to the game provider and/or honest game players. The nature of this type of applications poses a particular challenge to ensure cyber security because it is difficult to ensure high availability while preserving the integrity of the operation of these applications (Arkin et al., 1999; Viega & McGraw, 2002; Young & Yung, 2004; Zhao, 2007; Zhao, 2008; Zhang et al., 2011).

Byzantine fault tolerance (Castro & Liskov, 2002) is a well-known technique to achieve cyber security (Zhao, 2014). The technique aims to tolerate various malicious attacks to online systems by employing state machine replication (Schneider, 1990). However, as we mentioned earlier, Byzantine fault tolerance cannot be used as it is because it is not equipped with built-in solution to resolve the conflict of replication determinism requirement and the intrinsic randomness of the server operation. In this article, we elaborate how we address this dilemma using an online poker game application as a running example. In this application, a pseudo-random number generator (PRNG) is used to generate the pseudo-random numbers for shuffling the cards. We present two alternative strategies to cope with the intrinsic application nondeterminism. One depends on a Byzantine consensus algorithm and the other depends on a threshold signature scheme. Furthermore, we thoroughly discuss the strength and weaknesses of these two schemes.

BACKGROUND

In this section, we provide a brief introduction of PRNG, the entropy concept, and the methods to collect and enhance entropy.

A PRNG is a computer algorithm used to produce a sequence of pseudo-random numbers. It must be initialized by a seed number and can be reseeded prior to each run. The numbers produced by a PRNG are not truly random because computer programs are in fact deterministic machines. Given the same seed, a PRNG will generate the same sequence of numbers. Consequently, if an adversary knows the seed to a PRNG, then he/she can generate and predict the entire stream of random numbers (Young & Yung, 2004). Therefore, to make the random numbers unpredictable,

DOI: 10.4018/978-1-5225-2255-3.ch143

it is important that the seeds to the PRNG cannot be guessed or estimated. Ideally, a highly random number that is unpredictable and infeasible to be computed is required to seed the PRNG in order to produce a sequence of random numbers.

The activity of collecting truly random numbers is referred to as "collecting entropy" by cryptographers (Young & Yung, 2004). Entropy is a measure of the degree of randomness in a piece of data. As an example, consider using the outcome of coin flipping as 1 bit of entropy. If the coin-toss is perfectly fair, then the bit should have an equal chance of being a 0 or a 1. In such a case, we have a perfect 1 bit of entropy. If the coin-toss is slightly biased toward either head or tail, then we have something less than 1 bit of entropy. Entropy is what we really want when we talk about generating numbers that cannot be guessed. In general, it is often difficult to figure out how much entropy we have, and it is usually difficult to generate a lot of it in a short amount of time.

It is a common practice to seed a PRNG with the current timestamp. Unfortunately, this is not a sound approach to preserve the integrity of the system, as described by Arkin et al (1999) in the context of how a Texas Hold'em Poker online game can be attacked. They show that with the knowledge of the first few cards, they can estimate the seed to the PRNG and subsequently predict all the remaining cards.

TECHNIQUES FOR ENHANCING THE TRUSTWORTHINESS

In this section, we describe two possible strategies for enhancing the trustworthiness of online gaming applications. One depends on a Byzantine consensus algorithm and the other depends on a threshold signature algorithm. Both algorithms ensure that all replicas adopt the same value to seed their PRNGs, while each replica is taking entropy from its respective entropy source.

Byzantine Fault Tolerance

A Byzantine fault (Lamport, Shostak, & Pease, 1982) is a fault that might bring a service down, or compromise the integrity of a service. A Byzantine faulty replica may use all kinds of methods to disrupt the normal operation of a system. In particular, it might propagate conflicting information to other replicas. To tolerate f Byzantine faulty replicas in an asynchronous environment, we need to have at least 3f+1 number of replicas (Castro & Liskov, 2002). An asynchronous environment is one that has no bound on processing times, communication delays, and clock skews. Internet applications are often modeled as asynchronous systems. Usually, one replica is designated as the primary and the remaining ones as backups.

Any robust Byzantine fault tolerance (BFT) algorithm can be modified to cope with the use of random numbers. In the following, we describe a solution based on the well-known Practical BFT (PBFT in short) algorithm developed by Castro and Liskov (2002). The PBFT algorithm has three communication phases in normal operation. During the first phase, the pre-prepare phase, upon receiving a request from the client, the primary assigns a sequence number and the current view number to the message and multicasts a Pre-Prepare message to all backups. In the second phase, referred to as the prepare phase, a backup broadcasts a Prepare message to the rest of replicas after it accepts the Pre-Prepare message. Each non-faulty replica enters into the commit phase, i.e., the third phase, only if it receives 2f Prepare messages (from other replicas) that have the same view number and sequence number as the Pre-Prepare message, then it broadcasts the Commit message to all replicas including the primary. A replica commits the corresponding request after it receives 2f matching commit messages from other replicas. To prevent a faulty primary from intentionally delaying a message, the client starts a timer after it sends out a request and waits for f+1 consistent responses from different replicas. Due to the assumption that at most f replicas can