A SOA Approach to Counter DDoS Attacks

Xinfeng Ye Santokh Singh
Department of Computer Science, Auckland University, New Zealand
xinfeng@cs.auckland.ac.nz santokh@cs.auckland.ac.nz

Abstract

Distributed denial-of-service (DDoS) attacks are increasingly mounted by cyber-criminal gangs to extort money from online businesses. This kind of attacks is normally targeted at a particular service provider to exhaust the network and system resources of the provider. Since the scale of the attack is limited, the ISP operators normally cannot observe this type of attacks. As a result, the victim of the attack is left to deal with the attack on its own accord. This paper proposes a SOA approach to build a system against DDoS attacks targeting online businesses. The system is built on web services. It can be constructed and reconfigured easily by an attack victim. Experiments were also carried out to measure the overheads and the effectiveness of the proposed approach.

1. Introduction

A denial-of-service (DoS) attack is an attempt to make a computer resource (e.g. the network bandwidth, CPU time, etc.) unavailable to its intended users. To obtain the necessary network and CPU resources, attackers tend to use a large number of machines to launch Distributed DoS (DDoS) attacks. DDoS attacks can be devastating to the victims. Cyber-criminals are increasingly using DDoS attacks to extort money from online businesses [18, 24] or obtain commercial advantages [19]. To avoid detection, instead of using the traditional bandwidth flooding attacks, e.g. SYN flood, more and more attackers launch their attacks by mimicking legitimate Web access behavior of a large number of clients, e.g. running a large number of queries against the victim's site, downloading large image files from the victim's server. This type of attacks is called *CyberSlam* [11].

According to the scale of the attacks, DDoS attacks can be classified into *large-scale* and *medium-scale* attacks [13]. A large-scale attack causes widespread damage and affects a large portion of the network. The effect of a large-scale attack can be observed by the ISP operators. In a medium-scale attack, only a few servers or data centers are affected. As a result, the ISP operators might not be able to observe such an attack. [13] pointed out that the ISP operators care less about the medium-scale attacks as the effects of such attacks are not obvious to the ISP operators. As a result, the victims have to undertake the task of defending their servers against DDoS attacks.

Many solutions have been developed to prevent and trace DDoS attacks [3, 10, 22]. Many of these measures require wide-spread adoption to be successful. Unfortunately, there is a lack of incentive for the ISP operators to deploy these measures [2]. Due to the acuteness of the problems caused by DDoS attacks, some commercial tools, e.g. [15], are available for monitoring and analysing the network traffic. However, small or medium-sized businesses might not have the expertise to interpret and utilise the data generated by the tools. In this paper, a scheme for countering DDoS attacks is proposed. The scheme is based-on the SOA approach. It intends to help the small and medium-sized web services providers to cope with medium-scale CyberSlam attacks that are not observed by the ISP operators.

The paper is organised as follow. §2 presents our DDoS defense system. The performance evaluations of the system are given in §3. §4 discusses some features and possible extensions to the system. Related work is described in §5.Conclusions are given in §6.

2. A Scheme against DDoS Attacks

2.1. An Overview of the Scheme

A DDoS attack normally intends to exhaust the victim's network bandwidth and system resources (e.g. CPU, memory, etc.). An attacker can exhaust the network bandwidth of a victim by generating a large volume of traffic directed to the victim's site. The victim's system resources can be exhausted by processing the requests sent in by the attacker. The scheme in this paper avoids the network bandwidth of

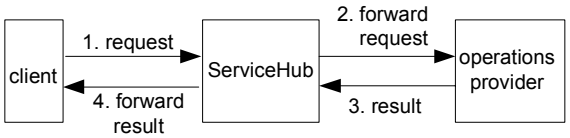
2007 IEEE International Conference on Web Services (ICWS 2007)
0-7695-2924-0/07 $25.00 © 2007 IEEE

the web services providers being exhausted by hiding the locations of the web services providers' sites from the public. To avoid system resources being exhausted by the attackers, authentication is used to verify the incoming requests before the requests are processed by the web services providers. In the following discussion, a web services provider is called an *operations provider*.

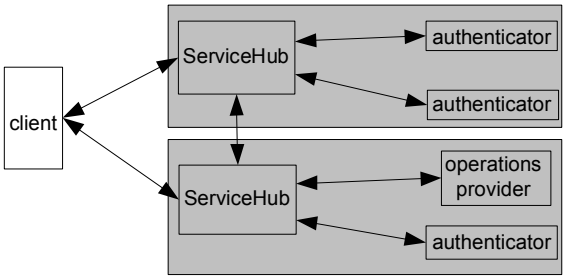
The scheme has two working modes, i.e. the normal mode and the under-attack mode. An operations provider decides which mode the system works in. When an operations provider does not detect any attack activity, i.e. the operations provider's system resources can cope with the service requests, the system works in the normal mode. Otherwise, the system works in the under-attack mode [1]. To minimise the delay in responding to users' requests, a service request is only authenticated when the system works in the under-attack mode. It is assumed that, in the service negotiation phase, an operations provider and its users have exchanged the necessary information that will be used in authenticating the requests.

It is assumed that a service provider with the expertise to handle security issues provides a service that forwards the requests from the clients to the operations providers that subscribe to the service. The site hosting the service is called the *ServiceHub*. It is assumed that the *ServiceHub* has sufficient network bandwidth to cope with medium-scale DDoS attacks[2]. Since the *ServiceHub*'s provider has the security expertise, the provider is able to use various techniques to trace and confine large-scale DDoS attacks. The WSDL file describing the operations (i.e. services) provided by the operations provider binds the operations to the *ServiceHub*. Thus, the public perceive the operations as being hosted by the *ServiceHub*. As a result, all service requests are first sent to the *ServiceHub*. Since an operations provider's address is unknown to the attackers, the attackers cannot send service requests directly to the operations provider. Thus, the attackers cannot easily exhaust the network bandwidth of the operations provider.

When working in the normal mode, the *ServiceHub* forwards the service requests to the operations provider directly. In the under-attack mode, the service requests need to be authenticated before being processed. The authentication verifies the identity of the sender of a service request. The operations provider will only process a service request if the identity of the client can be verified. In other words, the requests sent in by the attackers will be dropped. Therefore, the service provider does not waste system resources to process the attackers' requests. However, the authentication mechanism also uses system resources. An attacker can still deplete the victim's system resources by sending in a large amount of requests that force the victim to authenticate. To counter this kind of attack, an operations provider, say *op*, subscribes the services provided by other service providers to delegate the authentication task to the other service providers. The service providers that provide the authentication service are called *authenticators*. The authenticators only forward the requests that are successfully authenticated to *op*. Since the authentication is carried out by the other service providers, the attacker will not be able to exhaust *op*'s system resources.



(a) Normal Mode



(b) Under-attack Mode

Figure 1 A DDoS Defense System

Figure 1(a) shows how the system works in the normal mode. Service requests are first sent to the *ServiceHub* (step 1). The *ServiceHub* forwards the requests to the operations provider (step 2). The operations provider sends the results back to the client through the *ServiceHub* (step 3 and 4).

Figure 1(b) shows how the system works in the under-attack mode. The authenticators offer their services through their own *ServiceHubs*. A shaded area covers a *ServiceHub* and the service providers i.e. the operations provider and the authenticators, that subscribe to the *ServiceHub*'s service. As shown in the figure, it is possible that the authenticators and the operations provider use the same *ServiceHub*. The operations provider and the authenticators provide their services through their *ServiceHub*s. They send and receive messages through their *ServiceHub*s. Only their *ServiceHub*s know their locations. Thus, they cannot

[1] It is impossible to distinguish between DDoS attacks and legitimate peak time traffic. As can be seen later, the scheme does not discard any legitimate requests.

[2] As the *ServiceHub* provides the service to a large number of operations providers, the network bandwidth subscribed by the *ServiceHub* is likely to be much higher than the bandwidth subscribed by any individual operations provider.

exchange messages directly. In the under-attack mode, the operations provider informs its *ServiceHub* of the authentication services that it subscribes to. The *ServiceHub* will forward the service requests to the *ServiceHub*s of the respective authenticators. In turn, the authenticators' *ServiceHub*s will forward these messages to the corresponding authenticators to authenticate the requests. During the authentication process, the authenticators might need to exchange further messages relating to authentication information with the clients. If a service request is authenticated successfully, the authenticator sends the request to the operations provider through their respective *ServiceHub*s. After processing the request, the operations provider sends the result back to the client through its *ServiceHub*.

2.2. Assumptions about the System

It is assumed that the attacker may control an arbitrary number of machines that are widely distributed across the Internet. An attacker can sniff the packets sent out by the clients. The operations providers, the authenticators and the clients can exchange the keys for encrypting the authentication information using secured channels (e.g. using PKI [9]). An attacker does not know the encryption keys. An attacker cannot correlate the traffic going in and coming out from a *ServiceHub*. That is, an attacker cannot infer how the *SerivceHub* forwards an incoming service request[3]. It is assumed that a group of service providers have established trust with each other. They host the authentication services that can be used by each other to collectively defend against DDoS attacks. It is assumed that the clients interact with the system using the asynchronous callback pattern [4]. That is, a client sends a service request with a return address to the web service. The web service returns the result to the address specified by the client.

2.3. The ServiceHub

A *ServiceHub* might host many operations of different service providers. A *ServiceHub* does not distinguish between the operations providers and the authenticators. The authentication service provided by the authenticators are also regarded as operations provided by service providers. A *ServiceHub* maps the received service requests to their corresponding services providers. For each operations provider, the *ServiceHub* keeps an *authenticators table*. The table records the authentication services subscribed by the corresponding operations provider. In the table, an authenticator is identified by the service that it provides, e.g. *servicehub1.com/AuthenticationService.asmx*.

When an operations provider, say *op*, changes to the under-attack mode, it informs its *ServiceHub* by calling a web method of the *ServiceHub*. In the call to the web method, *op* passes a list of authentication services that will help *op* to authenticate the incoming requests. The *ServiceHub* creates an *authenticators table* for *op* and adds the authenticators to the table.

When a service request arrives, according to the operation to be invoked by the request, the *ServiceHub* finds the corresponding operations provider. If the operations provider does not have an *authenticators table*, it means that the operations provider is working in the normal mode. In this case, the *ServiceHub* forwards the request to the corresponding operations provider. Otherwise, the operations provider is working in the under-attack mode. In this case, the *ServiceHub* checks to see whether the request comes from an authenticator. If the request comes from an authenticator, the request must have been authenticated. Thus, it is forwarded to the corresponding operations provider. Otherwise, the *ServiceHub* randomly selects an authentication service from the table and forwards the request to the selected authentication service.

2.4. The Authentication Mechanism

The authentication mechanism relies on the symmetric key encryption algorithm [21]. At the service negotiation stage, a client and an operations provider have agreed on a shared key, $KEY_{client\text{-}provider}$, to encrypt the authentication information. A client can only encrypt the information that can be correctly decrypted by the authenticators if the client has the appropriate shared key. Thus, the possession of the shared key is the proof of the identity of the client.

When a request is authenticated, if no authentication information is attached to the request, a challenge string will be sent from the authenticator to the client. A challenge string consists of the id of the authenticator that created the string (*ID*), a sequence number (*SEQ_NUM*), the creation time of the string (*C_TIME*), and, the digital signature of the authenticator ($SIG_{authenticator}$). One-way hash function SHA1 [20] is used to calculate the digest of the concatenation of *ID*, *SEQ_NUM* and *C_TIME*. $SIG_{authenticator}$ is obtained by encrypting the digest with

[3] This assumption is reasonable, since many service providers might subscribe to the service of the *ServiceHub*. As a result, many requests might arrive at the *ServiceHub* simultaneously. Thus, the order in which these requests are forwarded can be non-deterministic.

key $KEY_{provider}$. $KEY_{provider}$ is only known to the operations provider and the authenticators. The authenticators are ordinary service providers. They might not have the expertise to cope with medium-scale CyberSlam attacks. Thus, it is important that the IP address of an authenticator is not known by the attacker. The *ID* of an authenticator in the challenge string is an integer assigned to the authenticator by the operations provider. The *ID* cannot be used to infer the IP address of the authenticator. Thus, the attacker cannot find out the location of the authenticator according to the *ID*. The operations provider informs the authenticator of its *ID* and $KEY_{provider}$ when the operations provider switches to the under-attack mode. When a client receives a challenge string, the client calculates the digest of the concatenation of *ID*, *SEQ_NUM* and *C_TIME* in the challenge string. The digest is encrypted using key $KEY_{client\text{-}provider}$ to form SIG_{client}. The challenge string and SIG_{client} form an *authentication token*. As the challenge string causes the creation of the token, the sender of the challenge string is called the *originator* of the authentication token. The sequence number (*SEQ_NUM*) in the challenge string is also called the *sequence number* of the token. Then, the client re-sends the original service request with the authentication token attached. The token is stored in the header element of the SOAP message that encapsulates the service request.

When changing to the under-attack mode, the operations provider informs the authenticators (a) the key used to sign the digest of the challenge string, i.e. $KEY_{provider}$, (b) a service list indicating the IDs assigned to the authenticators, (c) the secrete key $KEY_{client\text{-}provider}$ used by the client to generate SIG_{client}, and, (d) some other information that will be explained later. This information is passed to the authenticators through the web operations of the authenticators.

When a request is received by an authenticator, the authenticator first checks to see whether the request has an authentication token attached. If the request does not contain an authentication token, the authenticator sends a challenge string back to the sender of the request. If the authentication token exists in the request, the authenticator (a) uses $KEY_{provider}$ to decrypt $SIG_{authenticator}$ to obtain $\{SIG_{authenticator}\}^{-1}_{KEY_{provider}}$ ($\{M\}^{-1}_{KEY}$ means message *M* is decrypted using key *KEY*), and, (b) uses $KEY_{client\text{-}provider}$ to decrypt SIG_{client} to obtain $\{SIG_{client}\}^{-1}_{KEY_{client-provider}}$. The authenticator computes the digest of the concatenation of *ID*, *SEQ_NUM* and *C_TIME*. If (a) the digest is not equal to $\{SIG_{authenticator}\}^{-1}_{KEY_{provider}}$ or (b) $\{SIG_{client}\}^{-1}_{KEY_{client-provider}}$ is not equal to $\{SIG_{authenticator}\}^{-1}_{KEY_{provider}}$, the request fails the authentication check. Comparing the digest with $\{SIG_{authenticator}\}^{-1}_{KEY_{provider}}$ is to ensure that *ID*, *SEQ_NUM* and *C_TIME* in the token have not been tampered with, since they will also be used to check the validity of the token. Since *ID*, *SEQ_NUM*, and, *C_TIME* are sent in clear text in the challenge string, the attacker can easily compute the digest. However, the attacker does not know $KEY_{client\text{-}provider}$. Hence, the attacker cannot generate SIG_{client} correctly. As a result, the request sent in by the attacker will fail the authentication check. The authenticator drops the messages that fail the validation check.

A client might send multiple service requests to the same operations provider. In this case, the authentication token created for the first service request can be used in the subsequent requests. This avoids the need for the authenticators to send challenge strings in the subsequent requests. Since the subsequent requests can be authenticated immediately without the need of exchanging the challenge string and the authentication token, the response time to the subsequent requests can be shortened.

An attacker might intercept a service request with a valid authentication token and replay the intercepted message from multiple attacking sites. To counter this kind of attack, two measures are taken. The first measure is that an authentication token is only valid for a limited period of time. The length of the period is determined by the operations provider. The authenticators are made aware of the length of the period by the operations provider. This measure avoids a token being used by an attacker for an un-limited amount of time. If an authentication token attached to a request has expired, the authenticators will send a new challenge string to the client. Thus, if a client sends multiple service requests to an operations provider over a long period of time, the client might receive several challenge strings and needs to generate several authentication tokens accordingly. To check whether an authentication token has expired, an authenticator checks whether *C_TIME* in the token is within a certain time interval. *C_TIME* is the time that the originator of the token generates the challenge string. Since a *ServiceHub* forwards a service request to a randomly selected authenticator, an authentication token might not be authenticated by its originator. In order to check whether a token has expired, *C_TIME* should be normalised (e.g. using the Greenwich Mean Time). This is because the authenticators might locate in different time zones.

The other measure is to limit the number of times that an authentication token can be used for validating

requests. This measure ensures that, even if an attacker uses a token that has not expired, only a limited number of the attacker's requests can pass the authentication check. The originator of the token is responsible for maintaining the count on the number of times that the token has been used to validate the service requests. Each authenticator has an *authentication token usage table* (*ATU* table). After sending out a challenge string, the authenticator stores the sequence number (*SEQ_NUM*) of the string in the *ATU* table. The table also has a *usage count* column for recording the number of times that a token with the given sequence number has been used to authenticate service requests (i.e. the number of times that the token is attached to service requests). When an authenticator is asked to check the usage of an authentication token, the originator of the token increments the token's usage count in *ATU*. When the usage count of an entry reaches the usage limit, the entry is removed from the table. As a result, any subsequent request for checking the usage of the token will get a "not found" response.

After the authenticator verified that an authentication token has not expired, the authenticator checks with the originator of the token to see whether the token has reached its usage limit. The ID of the originator can be retrieved from the token. From the ID, the authenticator can find out the service provided by the originator according to the service list received from the operations provider. The authenticator will call the service to carry out the check. If the token has reached its usage limit, the authenticator will not forward the request to the operations provider. Instead, the authenticator sends a new challenge string to the client asking the client to provide a new authentication token. If the token has not exceeded its usage limit, the authenticator forwards the request to the operations provider's *ServiceHub* to allow the request to be delivered to the operations provider[4]. The authenticator includes its service address in the header of the SOAP message encapsulating the service request. This is to allow the *ServiceHub* of the operations provider to know that the request has been authenticated. Thus, the request will be forwarded to the operations provider instead of being forwarded to an authenticator.

2.5. Accessing an Operations Provider

The client program that calls the services of the operations provider needs to handle the interactions with the authenticators to provide the necessary authentication information. A C# assembly that includes classes for handling the interactions is provided. With these classes, the programmers do not need to write the code for handling the interactions with the authenticator. The most frequently used class is *ClientProxy*. The following example shows how a client program uses the *ClientProxy* class to call the service provided by the operations provider.

```
1  ClientProxy clientProxy =
   new ClientProxy(ws_address, op, parameter, handler,
                   handler_method, port, key_file);
2  clientProxy.SendRequest();
```

In line 1, *ws_address* is the address of the service to be called. *op* is the name of the operation. *parameter* is the value to be passed to the web service. *handler* is the class for handling the response from the call. *handler_method* is the method that contains the code for processing the result returned from the web service. *port* is the location where the response from the web service is returned. *key_file* is the file in which key $KEY_{client\text{-}provider}$ is stored. When the *SendRequest* method is called (line 2), *clientProxy* calls the service at *ws_address*. It also creates a thread that waits for response arriving at port *port*. When a response is received, the thread checks to see whether the response is the result returned from the operations provider or a challenge string sent by an authenticator. If the response is the result from the operations provider, the thread activates *handler_method* and passes the result to *handler_method*. Otherwise, the thread will generate the authentication token as discussed in §2.4 and resends the request attached with the authentication token to the service at address *ws_address*. The authentication token will be stored in an instance variable of the *clientProxy* object. If the *SendRequest* method of the *clientProxy* object is called repeatedly, the same authentication token will be used in the request being sent to the operations provider until a new challenge string is received. Class *ClientProxy* has a method *SetParameter* for setting the value to be passed to the web service.

Calls to web services are made by client programs. Thus, key $KEY_{client\text{-}provider}$ has to be made accessible to the client program. Since users might run the program as a background task, the user might not be available for entering the password protecting the encryption key when the authentication token needs to be generated. Hence, the key is stored in clear text in file *key_file*. Storing the key in clear text poses a security risk as the attacker might obtain the key by compromising the client's machine. However, since the key is only valid for the period during which the client wants to use the services of the provider, the risk is confined to this

[4] It is possible that some of the requests replayed by an attacker will reach the operations provider. This problem can be addressed by setting the usage limit to 1. That is, each request needs to provide its own authentication token.

period. Thus, if the shared key is compromised, the compromised key will not cause any security problem after the given period.

3. Evaluation

Experiments were carried out to evaluate the performance of the system. The experiments were carried out on a group of PCs connected by a 100Mbps Ethernet. Each PC is a HP workstation xw4200 with a 3.4GHz Pentium 4 processor and 2GB memory. All the programs are implemented using C#. The experiments assume that the *ServiceHub*, the operations provider, and, the authenticators are connected through a WAN. It is assumed that the one-way transmission delay over the WAN between a client and the other entities in the system is 60ms. It is also assumed that the service providers subscribe to the service of their local *ServiceHub*. As the *ServiceHub*, the authenticators and the operations provider are close to each other, the transmission delay amongst them is set to 15ms. To simulate the transmission delay, a program sleeps for 60ms or 15ms before it sends out a message.

In the following discussion, *arrival interval* means the interval between the clients' requests are sent to the system while *processing time* is the duration to process a client's request by the operations provider. In the experiment, it is assumed that the system has sufficient processing capacity to handle the legitimate clients' requests. Thus, the arrival interval of the legitimate clients' requests is set equal to the processing time of the operations provider.

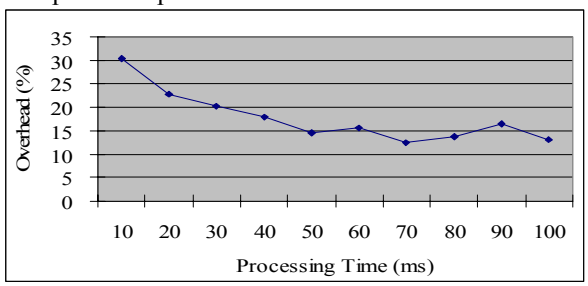


Figure 2 The Overheads of the *ServiceHub*

The first experiment measures the overhead of the system when the operations provider is working in the normal mode. The overhead is defined as $(t_{forward} - t_{direct}) / t_{direct}$ where t_{direct} is the service response time when the client's request is sent directly from the client to the operations provider (i.e. the *ServiceHub* is not used), and, $t_{forward}$ is the service response time when the client's request is forwarded to the operations provider by the *ServiceHub*. Figure 2 shows the results of the experiments. It can be seen that, as the processing time increases, the overhead of the system decreases. This is because, as the processing time increases, the forwarding cost for per unit of computation decreases. It can be seen that, when the processing time is longer than 40ms, the overheads is about 15%. Thus, the forwarding overhead of the system working in the normal mode is relatively low.

The second experiment measures the effectiveness of the system in countering DDoS attack. In this experiment, the system consists of one *ServiceHub*, one operations provider, and, two authenticators. The operations provider and the authenticators use the same *ServiceHub*. It is assumed that, (a) the operations provider and the authenticators use the same *ServiceHub*, (b) each authentication token can only be used to validate one service request, and, (c) the attacker sends requests to the operations provider at the rate of 1000 requests per second. The response time of the operations provider when the DDoS defense system is used (denoted as *with-defense*) and the response time of the system when the DDoS defense system is not used[5] (denoted as *no-defense*) are compared. Figure 3 shows the results of the experiments. From the figure, it can be seen that the responses time of the no-defense system increases as the processing time increases. This is because the operations provider processes each of the received requests (including the ones sent in by the attacker). Thus, as the processing time increases, the response time also increases. It also can be seen that, as the processing time increases, the response time of the with-defense system gradually decreases and becomes stable at around 1200ms. This is because, in our implementation, a thread is created to handle each of the received messages. When the arrival interval is short, too many messages arrive at the *ServiceHub* at the same time. The operating system of the *ServiceHub* only allows a limited number of threads to be created by a process at a time. Thus, sometimes the programs have to try multiple times in order to connect to the *ServiceHub* successfully to send the messages. The re-tries cause extra delay in processing legitimate users requests. When the arrival interval increases, the number of re-tries decreases. Since (a) the operations provider only processes the legitimate users' requests, (b) the legitimate users' requests arrive at the same rate as the operations provider's processing rate, and, (c) the time for exchanging the messages relating to authenticating a service request dominates the response time, the response time of the with-defense system becomes stable when the processing time is greater than 60ms. From Figure 3, it can be seen that,

[5] That is, the legitimate users and the attacker send requests directly to the operations provider.

compared to an un-protected system (i.e. the no-defense system), a system using the DDoS defense scheme proposed in this paper can significantly reduce the response time when the system is under attack.

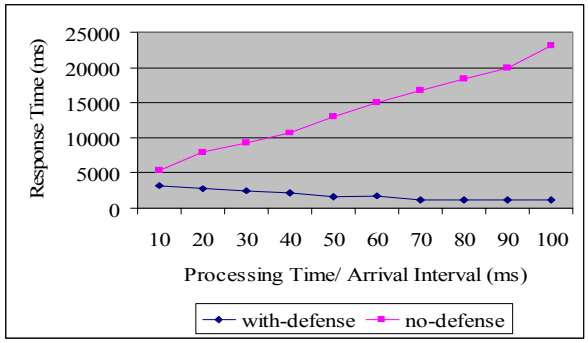


Figure 3 The Effectiveness of the DDoS Defense System

4. Discussions

Some operations providers might have some authentication mechanisms built-in to their services. That is, in order to invoke a service, a client needs to include some authentication information in the service request. The authentication mechanism used in this paper is orthogonal to the authentication mechanisms built-in to the operations providers' services. The reason for separating the two authentication mechanisms is due to flexibility concern. If the two authentication mechanisms are the same, then the authenticators must know how the operations provider implements its authentication mechanism. This means an authenticator needs to know the authentication mechanisms of all its potential. Obviously, this requirement is not practical.

Although the scheme in this paper focuses on CyberSlam attack, the architecture of the scheme can be used to cope with XML Denial of Service (XDoS) attacks discussed in [8]. In XDoS attacks, the attackers crash servers with incorrect data. To counter this kind of attacks, the authenticators can be used to validate the data being sent to the operations providers. Thus, it can be guaranteed that the data conform to the required format when they reach the operations providers.

Normally, an operations provider needs to pay the authenticators for using their services. To minimise the cost to the operations provider, depending on the scale of the attack, the operations provider can increase/decrease the number of the authenticators that are used to authenticate the incoming requests. The authenticators periodically report to the operations provider the number of requests being processed. According to the reported number, the operations provider can decide whether more or fewer authenticators will be used. The operations provider can start/stop using an authenticator by asking the *ServiceHub* to add/remove the authenticator from the operations provider's authenticators table.

The scheme in this paper assumes that an attacker cannot correlate the incoming and outgoing messages of the *ServiceHub*s. This assumption prevents the attackers from discovering the locations of the operations providers and the authenticators easily. However, this assumption is only reasonable if the traffic volume around a *ServiceHub* is high. To make this assumption hold when the traffic volumn around a *ServiceHub* is low, the scheme in [6] can be used. This means the *ServiceHub*s in the system form an anonymous overlay system. Encrypted forwarding tunnels are used to pass messages in the overlay. Fake traffic between the *ServiceHub*s are added to make it difficult to determine where traffic is actually originating and going.

The authentication mechanism can be extended to use PKI authentication to authenticate the users. This would allow users who have not completed service negotiation with the operations provider (i.e. the users and the operations provider have not agreed on key $KEY_{client\text{-}provider}$) to be validated when the system is working in the under-attack mode.

5. Related Work

Andersen et al. [2] and Keromytis et al. [12] discuss using a secure overlay to filter and route the messages being sent to a destination. In [2] and [12], the location of the destination of a message is unknown to the sender of the message. The sender of the message sends the message to an overlay. The nodes in the overlay verify the validity of the message and forward the message toward the destination of the message. The scheme in this paper uses similar idea as [2] and [12] in terms of hiding the locations of the service providers from the public. However, the schemes in [2] and [12] need cooperation from the ISP operators to modify the DNS entries when an attack occurs. The overlay network in [2] and [12] cannot be reconfigured easily by the message receivers at run time. In contrast, the scheme in this paper does not require the ISP operators to be involved when the system switches to the under-attack mode. Also, the operations providers can dynamically decrease or increase the number of authenticators in the system. Thus, the system in this paper is more flexible and easy to use.

Padmanabhuni et al. [17] proposed a framework for detecting and preventing DoS attacks against web services. The scheme focused on validating XML messages efficiently. In [17], users need to authenticate with the system. Since their scheme handles the

validation at one site, unlike our scheme, [17] is not suitable for countering the CyberSlam type attacks.

Some schemes [7, 11, 16] require human attention when authenticating the users. For example, [11] provides an operating system kernel extension to protect Web servers against DDoS attacks. It uses a reverse Turing test [23] to authenticate users. Web Services are normally accessed through programs. The reverse Turing test cannot be carried out by the programs easily. Thus, the scheme proposed in [23] cannot be directly applied in a SOA environment.

There are schemes using computational puzzles that require the client to carry out heavy computation before accessing the services [1, 5, 14]. The computation would slow down the attacker. However, this kind of scheme is not effective against DDoS attacks. This is because the attacker can utilise a large number of machines. Thus, it is unlikely that the attacker will be short of computing power.

6. Conclusion

This paper proposes a scheme for forming a DDoS attacks defense system in a SOA environment. A service provider can construct such a system by subscribing the services provided by other providers. With this system, a service provider does not need to rely on the ISP operators to provide solutions for countering medium-scale CyberSlam type attacks. Since the system is constructed from web services, it can be formed and reconfigured easily. The empirical data shows that, when there is no DDoS attacks (i.e. when the system is working in the normal mode), the overheads of the system seems to be acceptable. The experiments also show that, when the system is under attack, the response time to the legitimate users is much shorter than the system that is not protected by the scheme proposed in this paper. Thus, the proposed system is effective in countering DDoS attacks.

Acknowledgment. We would like to thank the anonymous reviewers for their useful comments.

References

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. ACM Trans. on Internet Technology, Vol. 5, Issue 2, 2005, pp. 299 - 327

[2] D. Andersen. Mayday: Distributed Filtering for Internet services. Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS), 2003

[3] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In Proc. of Hotnets-II, Cambridge, MA, Nov. 2003

[4] M. Brambilla, S. Ceri, M. Passamani, A. Riccio: Managing Asynchronous Web Services Interactions, Proc. of the IEEE Intl Conf on Web Services, 2004

[5] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In Advances in Cryptology - CRYPTO 2003. LNCS 2729, pp.426-444

[6] M.J. Freedman & R. Morris, Tarzan: A peer-to-peer anonymizing network layer, Proc. of the 9th ACM Conf. on Computer and Communications Security, 2002

[7] V. D. Gligor. Guaranteeing access in spite of distributed service-flooding attacks, in Proceedings of the Security Protocols Workshop, April 2003

[8] B. Hartman, Securing your Enterprise Web Services in a Suspicious world, SOA Web Services Journal, March 2004, Available at http://webservices.sys-con.com/read/43958.htm

[9] R. Housley, W. Polk, W. Ford, and D. Solo, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, IETF RFC3280

[10] J. Ioannidis and S. M. Bellovin, Implementing pushback: Router-based defense against DDoS attacks, Proc. of Network and Dist. System Security Symposium, 2002

[11] S. Kandula, D. Katabi M. Jacob and A. Berger, Botz4Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds, Proceedings of the 2nd Symposium on Networked Systems Design and Implementation, 2005

[12] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: An architecture for mitigating DDoS attacks. Journal on Selected Areas in Communications, 21(1):176–188, 2004

[13] E. Kohler, Denial of Service Defense in Practice and Theory, USENIX'05, http://www.usenix.org/event/usenix05-/tech/slides/kohler.pdf

[14] D. Mankins, R. Krishnan, C. Boyd, J. Zao, and M. Frentz. Mitigating distributed denial of service attacks with dynamic resource pricing. Proc. IEEE Annual Computer Security Applications Conference, Dec. 2001.

[15] Mazu Network, Mazu Profiler, http://www.mazunet-works.com/solutions/

[16] W. Morein and A. Stavrou and D. Cook and A. Keromytis and V. Misra and D. Rubenstein, Using Graphic Turing Tests to Counter Automated DDoS Attacks Against Web Servers, Proc. of the 10th ACM Intl. Conf. on Computer and Comm. Security (CCS), pp. 8-19, 2003

[17] S. Padmanabhuni, V. Singh, K. M. S. Kumar, and A. Chatterjee, Preventing Service Oriented Denial of Service (PreSODoS): A Proposed Approach, Proc. of the IEEE Intl. Conference on Web Services (ICWS'06), pp577 – 584, 2006

[18] D. Pappalardo and E. Messmer, Extortion via DDoS on the rise, NetworkWorld, May 2005. http://www.network-world.com/news/2005/051605-ddos-extortion.html

[19] K. Poulsen. FBI Busts Alleged DDoS Mafia, 2004. http://www.securityfocus.com/news/9411.

[20] RFC 3174, US Secure Hash Algorithm 1 (SHA-1), http://tools.ietf.org/html/rfc3174

[21] B. Schneier, Applied Cryptography,Second Edition, John Wiley & Sons, 1996

[22] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, B.Schwartz, S. T. Kent and W. T. Strayer, Single-packet IP traceback, IEEE/ACM Transactions on Networking (TON) Volume 10 , Issue 6, 2002, pp. 721 - 734

[23] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *CACM*, 47(2), Feb. 2004.

[24] M. Ward, Caught in the net, http://news.bbc.co.uk/2/hi/technology/5407508.stm