A Language-Based Approach to Prevent DDoS Attacks in Distributed Object Systems *

Toktam Ramezanifarkhani, Elahe Fazeldehkordi, and Olaf Owe
Department of Informatics, University of Oslo, Norway

Abstract

Denial of Service (DoS) attacks and Distributed DoS (DDoS) attacks with higher severity are historically considered as one of the major security threats and among the hardest security challenges. Although there are lots of defense mechanisms to overcome such attacks, they are making the headlines frequently and have become the hugest cyberattacks, recently in 2016 and 2017. In this paper, our aim is to show how distributed program analysis can help to combat these attacks as an additional layer of defense. We consider a high-level imperative and object-oriented framework based on the actor model with support of asynchronous and synchronous method interaction, and shared futures, which are sophisticated features applied in many systems today. Since the preceding step in these attacks is flooding, we show how such communication can cause flooding and thus DoS or DDoS. Then, we provide a hybrid approach including the static and dynamic phases in distributed systems to prevent these attacks statically and to detect them at runtime based on the inline monitoring.

Introduction

Denial of Service (DoS) attacks are becoming crucial. Moreover, *Distributed DoS* (DDoS) attacks have even higher severity and the worst DDoS attacks happened (multiple times) in 2016 and 2017 [3]. More than 90 reports in the first month of 2017 were about DoS attacks. Recent DDoS attacks have imposed high financial overhead as well. Since 70 percent of the exploited devices are unmanaged and have weaknesses, and since there are tens of millions of such devices out there, we face a huge problem, and thus it is inevitable that applications in such devices can be used as bot-nets again. Although there are lots of proposed defense mechanisms to overcome these attacks [1, 2]such as packet filtering or intrusion detection systems, based on the recent experiences, they are not enough and it is required to strengthen them. Moreover, existing bots are likely to live and they are not going away for a while.

In our setting and underlying language, due to some sophisticated features such as asynchronous and non-blocking method calls, it is even easier for the attacker to launch a DoS, because then undesirable waiting by the attacker is avoided in the distributed setting. Therefore, we adapt a static technique to prevent flooding and thus DOS attacks. Moreover, instrument the code for dynamically checking of probable attacks to prevent them at runtime. By including the static analysis in the compilation phase, one obtains static and automatic built-in DoS prevention, and dynamic DoS detection at runtime. In this paper we consider a high-level imperative and object-oriented language based on the actor model with support of asynchronous and synchronous method interaction. We explain our hybrid approach including the static and dynamic phases in this model of distributed systems, and show some examples.

Static and Dynamic Attack Detection and Prevention: To launch a DoS attack, the attacker tries to submerge the target server under many requests to saturate its computing resources. To do so, flooding attack by method calls are effective especially when the server allocates a lot of resources in response to a single request. So, we detect

- call-flooding: flooding from one object to another, which is similar to GET-based flooding, and

*Work supported by the SCOTT and IoTSec (Norwegian Research Council) projects. SCOTT (www.scott-project.eu) has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, Norway.

- parametric-call-flooding: flooding from one object to another when the target object allocates resources or consume resources for each call.

For any set of methods that call the same target method, a call cycle could be harmful. The methods might belong to the same or different objects with the same or different interface. With the possibility of non-blocking calls, it is even more cost-beneficial for the attacker to launch a DoS, because then undesirable waiting by the attacker is avoided in the distributed setting. By means of futures and asynchronous calls, a caller process can make non-blocking method calls that we have considered in an example. This case can be detected statically, involving several factors:

- There should not be lots of methods that can call the same method, simultaneously. With respect to static detection, it is in general hard to see if the callee is the same for different calls. However, a category of self calls can be detected.
- Although we can not trace calls statically, for each target method we can automatically instrument a security code to check the number of calls it receives in a time frame, and block the callers as an anomaly detection and reaction. Moreover, to minimize the runtime overhead, we statically detect critical methods, such as those that are called as non-blocking or by the suspension method that are beneficial for the attackers and do the instrumentation for runtime detection.
- To prevent and detect parameterized DoS or DDoS attacks, the same static and dynamic approach is used while calls with parameters and resource allocations are considered as more serious situations.
- Since the possibility of infinite object creation as referred to as *instantiation flooding* could cause resource consumption and DoS which could be detected statically, especially if those objects and their communication can cause flooding requests in the bots such as clients in our example. Moreover, it is even worse if there is instantiation flooding at the target side of the distributed code. However, this can be detected by static analysis of the target.

Moreover, our anomaly detection is not based on source machine IP addresses that can be forged through a proxy or IP address spoofing. Therefore, for runtime anomaly detection it is possible to check the situations in which thousands of requests are coming to one object every single second specially when they have the same size or parameter settings which is common in automatic flooding attacks.

An Example of Instantiation Flooding: Fig. 1 (a) exploits unbounded creation of client objects where each client object is unaware of the attack. Interfaces are similar to those above and are not given. Each client is innocent in the sense that it does not cause any attack by itself. However, the attacker object makes an attack by using an unbounded number of clients to flood the same server s. The attacker does not wait for the connect calls to complete, therefore it is able to create more and more work load for s in almost no time. The execution of f=c!connect(s) causes an asynchronous call and assigns a future to the call. Thus no waiting is involved. It is immediately followed by a recursive asynchronous call, causing the current run execution to terminate before a new one is started. The attacker creates flooding by rapidly creating clients that each perform a resource-demanding operation on the same server.

Static Analysis of DoS Attacks: We apply the static analysis of flooding presented in [4] for detection of flooding of requests, formalized for the Creol setting. We adapt this notion of flooding to deal with detection of DDoS attacks, which have a similar nature. The static analysis will look for flooding cycles in the code. According to [4] flooding is defined as follows:

> An execution is *flooding with respect to a method* m if there is an execution cycle, call it C, containing a call statement $o!m(\overline{e})$ at a given program location, such that this statement may produce an unbounded number of uncompleted calls to method m, in which case we say that the call $o!m(\overline{e})$ is *flooding with respect to* C.

Flooding is detected by building the control flow graph of the program and locating control flow cycles as shown in Fig. 1 (b). Then, the sets of weakly reachable calls, denoted *calls*, and the set of strongly reachable call completions, denoted *comps*, in each cycle have to be analyzed. Flooding is reported for each cycle with a nonempty difference between *calls* and *comps*, as explained in Fig. 1 (c). Note that the abbreviated notations for synchronous calls and suspending calls are expanded to the more basic call primitives explained above.

Weakly reachable nodes are those that are reachable from the cycle by following a flow edge or a call edge. A node is strongly reachable if it is on the cycle or is reachable without passing a wait node (outside the cycle) unless the return node of the corresponding call is strongly reachable. Also nodes that lead to a strongly reachable

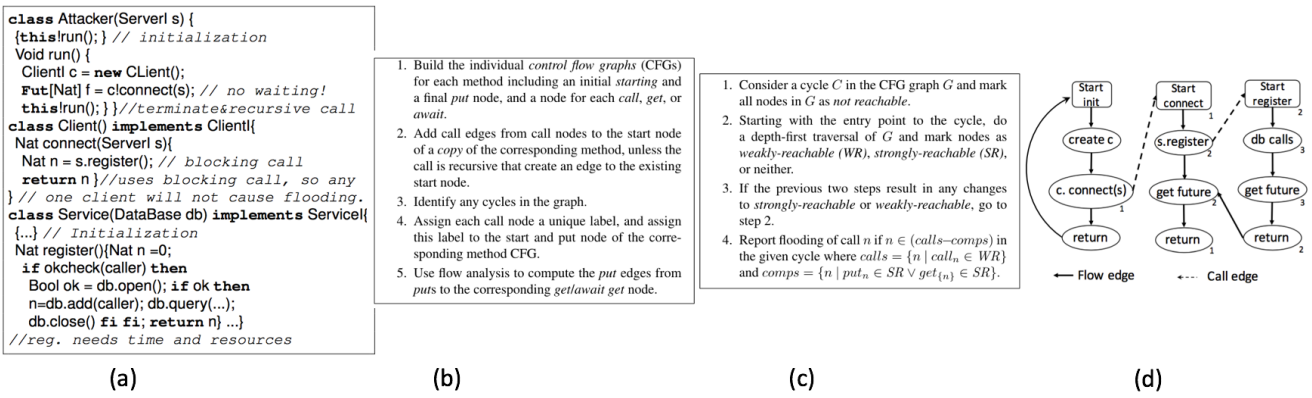

Figure 1. (a), Flooding by unbounded creation of innocent clients targeting the same server, (b) Static detection of flooding, (c) Control Flow Graph, and (d) Top Level algorithm for detecting flooding relative to a given cycle.

node without leaving an if-branch are also strongly reachable. A more precise detection is found in [4], which distinguishes between weak flooding and strong flooding. Strong flooding is flooding under the assumption of so-called *favorable* process scheduling. Strong flooding reflects the more serious flooding situations that persist regardless of the underlying scheduling policy. In the detection of strong flooding, an enabled node is considered strongly reachable if its predecessor flow node(s) are strongly reachable. With respect to DDoS, weak flooding of a server is in general harmless unless the flooding is caused by a large enough number of objects. And strong flooding is dangerous even from one single attacker. A server is often running on powerful hardware, even compared to that of an attacker, in which case it suffices to look for strong flooding. However, weak flooding may be used to discover botnet attacks, since in this case the combined speed of the attacker can be significantly higher than that of the attacked object.

Static Analysis of the Instantiation Example: For the creation of an attacker object, new Attacker(s), the following cycle is detected: - the initialization of the attacker calls *run*, *run* creates a client object *c*, *run* calls c!connect(s), *run* terminates and calls itself recursively. The *run* call has a call edge to the flow graph of *connect*, and *connect* has a call edge to the flow graph of *register*. The call to *register* waits for completion of *register* since it is a blocking call, and the database calls made by *register* wait for the completion of these database calls. The code for the database is not given, and therefore the analysis will be a worst case by considering such calls possibly non-terminating. The control flow graph is given in Fig. 1 (d) The weakly reachable call nodes of the cycle, i.e., *calls*, are $\{1, 2, 3\}$, and the strongly reachable calls, i.e., *comps*, are $\{1\}$. This gives that $calls - comps$ is $\{2, 3\}$. Thus call 2 gives a potential flooding, but in this case it does not reflect a real flooding since each instance of call 2 is on a separate object. This could be understood by (an extension of) the static checking analysis since it is on a new object generated in the same method. Furthermore, call 3 is detected as potentially flooding, and this reflects a real flooding situation. Thus the analysis detects the dangerous flooding of the server s, which is a DDoS attack.

References

[1] C. Douligeris and A. Mitrokotsa. DDoS attacks and defense mechanisms: classification and state-of-the-art. *Computer Networks*, 44(5):643–666, 2004.

[2] N. Gruschka and N. Luttenberger. Protecting web services from dos attacks by soap message validation. In *IFIP International Information Security Conference*, pages 171–182. Springer, 2006.

[3] New Mirai variant hits target with 54-hour DDoS, 2016. *https://www.infosecurity-magazine.com/news/new-mirai-variant-hits-target-with/*.

[4] O. Owe and C. McDowell. On detecting over-eager concurrency in asynchronously communicating concurrent object systems. *Journal of Logical and Algebraic Methods in Programming*, 90:158 – 175, 2017.