How to prevent CWE-770: Allocation of resources without limits

Nabeel Mahdi Abid Mehdi Sayed
2020-04-12

CWE 770 is one of the major causes of Denial of Service (DoS) attack on systems. This weakness arises due to improper management of resources and missing a security tactic during the design phase. In this paper we will discuss the design patterns and strategies which should be employed to mitigate the risk of CWE-770.

1. Introduction

CWE-770 is a child of CWE-400 (Uncontrolled Resource Consumption) and the MITRE corporation has described the weakness in this way,

"The software does not properly control the allocation and maintenance of a limited resource thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources." [MITRE770]

This weakness is one the major causes of Denial of Service (DoS) Attacks due to resource (CPU, memory, other) consumption, where when the resources necessary to perform an action are entirely consumed, the action is prevented from taking place. One good example of this would be CVE-2018-10908 [CVE10908] where a management module named VDSM was invoking a process named qemu-img on untrusted inputs without limiting resources. An attacker could cause the qemu-img process to consume unbounded amounts of memory of CPU time, causing a denial of service condition that could potentially impact other users of the host.

CWE-770 can be simply explained as a weakness which is caused when the system allocates resources without checking for enough conditions and ends up creating problems. The problems involved are usually centered towards exhaustion of resources and cause Denial of Service (DoS). To comprehend the cause of CWE-770 in a better way, let's go through a few examples.

1.1. Example 1

Below is an example [MITRE770] where a server object creates a server socket and accepts client connections to the socket. For every client connection made to a socket, server generates a thread object using ClientSocketThread class that handles request made by the client through the socket.

```
public void acceptConnections() {
try {
ServerSocket serverSocket = new ServerSocket(SERVER_PORT);
int counter = 0;
boolean hasConnections = true;
while (hasConnections) {
Socket client = serverSocket.accept();
Thread t = new Thread(new ClientSocketThread(client));
t.setName(client.getInetAddress().getHostName() + ":" + counter++);
t.start();}
serverSocket.close();
} catch (IOException ex) {...}
}
```

Over here, there is no limit to the number of client connections and client threads that can be created by the server. Allowing a huge number of client connections and threads could possibly deplete the system resources.

1.2. Example 2

In the following example, the processMessage function takes an input as a two-dimensional character array containing the message to be processed. The two-dimensional character array contains the length of the message in the first character array and the message body in the second character array. The getMessageLength function returns the integer value of the length from the first character array. After checking whether the message length is greater than zero, the body character array pointer points to the start of the second character array of the two-dimensional character array and memory is allocated for the new body character array [MITRE770].

```
int processMessage(char **message)
{
char *body;

int length = getMessageLength(message[0]);

if (length > 0) {
body = &message[1][0];
processMessageBody(body);
return(SUCCESS);
}
else {
printf("Unable to process message; invalid message length");
return(FAIL);
}
}
```

This example creates a situation where the length of the body character array can be very large and will consume excessive memory, exhausting system resources.

As we have observed, how CWE-770 is caused, we must now find ways to mitigate the risk of encountering such conditions and come up with a design which eliminates the possibility of encountering CWE-770.

2. Countermeasures

In this section we will be going through the countermeasures required to mitigate the risk of CWE-770. Although, there are many ways of mitigating CWE-770, we will be discussing the topics which are important from the point of view of design and implementation.

CWE-770 is a broad problem and requires a lot of design mechanisms to make the system secure from it. Even after employing the mechanisms, there is always a residual risk associated which can never be completely removed but it can be reduced by mitigating the smaller risks one by one. Each of these smaller risks can be removed by using some design tactic. The design and implementation mechanisms which should be employed to deal with those risks are listed below.

2.1. Define expected behavior

By defining expected behavior, we mean to clearly specify

- The minimum and maximum expectations for capabilities
- The behaviors which are acceptable when resource allocation reaches limits
- Behavior of system when an error occurs

Many of the vulnerabilities which have been caused by CWE-770 were because certain conditions were not checked. In the 'Example 2' of section 1.2, we saw that the program was not checking the length of the message, which could cause Denial of Service (DoS) if the attacker gave a message of huge length. By defining the expected behavior clearly, we can decrease the possibility of such vulnerabilities. Let's enumerate through the parts of a system which should be clearly defined to mitigate the risk of CWE-770.

2.1.1. Input Validation

In certain applications user's input influences the size or frequency of resource allocations. In such situations, an attacker can exhaust the resources by providing a malicious input.

2.1.1.1. Whitelist

We should always assume that all inputs are malicious and accept only those inputs which are deemed as good. One method of implementing this would be making of a list of all inputs which should be acceptable by the system and rejecting everything else. Such kind of a pattern where only some known inputs are accepted and everything else is rejected is known as a 'Whitelist'.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. [MITRE770]

2.1.1.2. Duplicated checks.

Attackers can easily bypass the security checks which exists on the client side by modifying the values after the checks have been performed and then he can send a malicious input to the server. This implies that security checks on the client side cannot be trusted and all inputs should be rechecked on the server side. Basically, all inputs should be checked at both client and server side.

2.1.2. Using logs correctly

Back in 2013, it was found that Linux kernel had an issue and a certain driver was flooding the kernel with messages which exhausted the resources and caused Denial of Service [CVE0231]. The driver was creating too many log messages and saving them in files. After a point the files became so huge that there was no storage space left to store them.

Writing log messages is a popular practice in programming and it helps the system in a long run. At the same time, logs can also be used as an exploit by an attacker. All logs get stored in a text file. If the logging is unrestricted, attackers can fill up the text file with a large number of log entries and crash the machine by filling up the storage. Once the entire hard drive is completely out of bytes, the machine can behave in unexpected ways and an attacker can take advantage of this.

We have found two mitigations of this problem [GITHUBCVE0213].

1. Use a properly configured logging library (e.g. log4j). In configuring your logger, be sure to use rolling log files. These can be rotated on a daily basis, or by size. Be sure to test this functionality yourself.
2. Don't log too much. Separate out the debugging logs from useful logs. Don't ship with debugging information turned on.

2.1.3. Fail gracefully

There used to be switches that were vulnerable to “macof” attacks. These attacks flooded a switch with random IP and MAC address combinations, therefore exhausting the switch’s cache, which held the information of which port corresponded to which MAC addresses. Once this cache was exhausted, the switch would fail in an insecure way and would begin to act simply as a hub, broadcasting all traffic on all ports and allowing for basic sniffing attacks [MITRECLASP2005].

The above incident is an example of a failure which compromised the security of a network. Failures will always happen but If the program must fail, it should be designed so that if the program does fail, the safest result should occur. When a failure happens there are two behaviors which the program can follow,

- Fail open: Try to fail in a way that allows system to function (choose if availability most important)
- Fail closed: Restrict access on failure, even if it means loss of service (choose if confidentiality or integrity are more important)

The programmer must determine whether fail open or fail close should be used in time of a failure. In most case, if some sort of failure is detected in a program then the program should immediately deny service and stop processing that request. Sometimes this can decrease reliability or usability, but it increases security. It is recommended to “stop processing the request”, not “fail altogether”. Although Sometimes taking the whole server down is necessary [Wheeler2004].

2.1.4. Resource handling

A lot of times improper handling of resource can create several problems including resource-exhaustion. There are two major problems which we will discuss in this section. The first one is memory allocation systems where improper allocation of memory causes the system to crash. The second one is associated with resource leakages. Here, the server fails to make it available after use. Examples are a component that neglects to close a file descriptor or to free some memory.

Let's discuss how these problems can be solved by using an efficient design tactic.

2.1.4.1. Do not create your own memory allocation system

Many programmers re-implement memory allocation system internally for speed (e.g., "slab allocators"). Although such implementation of programs is sometimes faster, it comes at a cost. Many defensive & testing systems modify/override default memory allocator, e.g., malloc() and new. If you create your own allocation system, these tools will fail to detect problems they'd otherwise detect. The defensive systems usually create guard pages to detect accesses beyond allocated region and limits the memory allocation to the limit defined. If memory is dynamically assigned, the program can go on a memory allocation and can also lead to buffer allow. [WheelerSlide3]

2.1.4.2. Prevent resource leaks

The resources of a machine are limited. Failing to use them efficiently can lead to denial-of-service, poor performance, & sometimes serious errors. Methods of mitigating resource leaks are listed below.

2.1.4.2.1. Finally /try-with-resources

Finally blocks (C++, Java, etc.) always execute after the try block irrespective of whether catch block was used or not. It is very useful for returning all the resources once the program is done using them. If you are using java, declare resources before and outside the try block and then in "finally" block, verify that the resource has been allocated/consumed, and if it has, deallocate/release. Beware that statements in "finally" may themselves throw exceptions. Example [WheelerSlide8],

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                                    throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Java provides another option for releasing resources; it is called try-with-resources. Java try-with-resources automatically releases system resources when no longer needed. We must ensure that object to be managed implements AutoCloseable for try-with-resources to work. [WheelerSlide8]

2.1.4.2.2. C++ constructor/destructor

Using a combination of constructor and destructor is a good method for properly releasing resources. A constructor is called when an object is created, and the destructor is called when object goes out of scope. Ensure all resources allocated in constructor are released in its corresponding destructor. Beware: If copying an object doesn't duplicate its resource, then destroying the original object and its copy will try to deallocate the same resource twice. If copying the object does duplicate the resource, that duplication is not going to happen by itself – you must write code to do that. [WheelerSlide8]

2.1.4.2.3. Incorrect error handling

Resource leaks are often caused by incorrect error handling. We must ensure that the application performs the appropriate error checks and error handling in case resources become unavailable. Two common mechanisms for checking whether an error has occurred are return codes & exceptions.

Return value of function/method may indicate that an error has occurred. Return values can be overloaded to include return value (if okay) and error code (if an unexpected condition takes place). For example, you can write a function which returns a number between 0 to INT_MAX when the function works as expected and return -1 when there is an error. Caller should always check every return value, and if the return value is of an error, caller should handle it. Another suggestion while handling errors in C would be to add error handling code at the end of the function. We can use 'goto' to jump to the end.

Nearly all languages have an "exception handling" mechanism. We must always use the "exception handling" methods to make our program secure. While writing code for catching exceptions, catch all exceptions. Be specific about the exceptions you catch. Don't just catch a generic exception, catch a subclass and be specific about it. Catch exceptions only when you can do something appropriate with it. Attackers will always try ways to trigger exceptions, due to which we should always the exception handlers are secure [WheelerSlide8].

There are few other handling mechanisms also available. It's the programmer's discretion, how does he want to handle the errors. The above two are just one of the many methods which can be used for error handling.

2.2. Limit or Throttle Resources

The most common solution of mitigating CWE-770 is to apply limits on the resources being used. By constraining the amount of resources for a single user or program, we can prevent resource exhaustion. In "Example 1" from section 1.1, the program was creating threads for the client. If the client sent too many requests, the program would create those many threads and might end up using all the CPU capacity. There are many ways by which these limits can be set, in this section we will take an overview few of such techniques. Along with limiting the resources, we will also take a glance at how throttling can be used to prevent Denial of Service (DoS).

2.2.1. Storage resource limits

We can define storage (filesystem) quota limits on each mountpoint for the number of blocks of storage and/or the number of unique files (inodes) that can be used, and you can set such limits for a given user or a given group. A “hard” quota limit is a never-to-exceed limit, while a “soft” quota can be temporarily exceeded [Wheeler2004].

In Linux, there are few commands which should be known to know how to use quota.

- ‘**quota’** displays users' disk usage and limits.
- ‘**quotaon**’ tells the system that disk quotas are enabled for a specific file system.
- ‘**quotaoff**’ tells the system that the specific file sytems should have their disk quota turned off.
- ‘**warnquota**’ checks the disk quota for each local filesystem and sends a warning message to those users who have reached their soft limit[LinuxWarnq].

This system of using quotas can be used to set per-user and per-group limits on the amount of disk space used on a file system. For each user and/or group, a soft limit and a hard limit can be set for each file system. The hard limit can't be exceeded. The soft limit can be exceeded, but warnings will ensue. Moreover, the user can't exceed the soft limit for more than one week (by default) at a time; after this time, the soft limit counts as a hard limit [LinuxQuotactl].

```
#include <sys/quota.h>
#include <xfs/xqm.h>

int quotactl(int cmd, const char *special, int id, caddr_t addr);
```

The quotactl() call manipulates disk quotas. The cmd argument indicates a command to be applied to the user or group ID specified in id. The special argument is a pointer to a null-terminated string containing the pathname of the (mounted) block special device for the file system being manipulated. The addr argument is the address of an optional, command-specific, data structure that is copied in or out of the system [LinuxQuotactl].

2.2.2. Process resource limits

Process memory is divided into several segments. Program code is contained in the text segment. The stack segment contains stack frames. The segment for stack grow to any size, though typical programs have bounded stacks. You can limit the stack size for a process with the use of stacksize rlimit. The data segment contains other writable data. The data segment can grow to any size. You can limit per-process data size with the datasize rlimit [BernsteinRlimit].

The rlimit mechanism supports a large number of process quotas, including file size, number of child processes, number of open files, and etc. There is a “soft” limit and a “hard limit”. The soft limit cannot be exceeded at any time, but through calls it can be raised up to the value of the hard limit [Wheeler2004].

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);

int prlimit(pid_t pid, int resource, const struct rlimit *new_limit,
struct rlimit *old_limit);
```

In the above example [LinuxPrlimit], the getrlimit() and setrlimit() system calls get and set resource limits respectively. Each resource has an associated soft and hard limit, as defined by the rlimit structure:

```
struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

The Linux-specific prlimit() system call combines and extends the functionality of setrlimit() and getrlimit(). It can be used to both set and get the resource limits of an arbitrary process. The resource argument has the same meaning as for setrlimit() and getrlimit() [LinuxPrlimit].

Other languages like python also provides the rlimit functionality. Here is a sample code on how rlimit can be used in python to restrict CPU time [GforgLimit].

```
# importing libraries
import signal
import resource
import os

# checking time limit exceed
def time_exceeded(signo, frame):
    print("Time's up !")
    raise SystemExit(1)

def set_max_runtime(seconds):
    # setting up the resource limit
    soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
    resource.setrlimit(resource.RLIMIT_CPU, (seconds, hard))
    signal.signal(signal.SIGXCPU, time_exceeded)

# max run time of 15 millisecond
if __name__ == '__main__':
    set_max_runtime(15)
    while True:
        pass
```

SIGXCPU signal is generated when the time expires on running this code and the program can clean up and exit.

2.2.3. Throttling resources

One of the most useful ways to deal with resource starvation is to code your application to change behavior based on whether it is under attack. Microsoft's SYN flood protection works the same way: if you have plenty of resources available, the system behaves normally. If resources are running low, it will start dropping clients who aren't active. The Microsoft file and print services "the Server Message Block (SMB) protocol and NetBIOS" use the same strategy [Micheal2003].

One-way throttling could be implemented can be in this way. An attack that doesn't require authentication is easier for an attacker. We can drop sessions that have failed to authenticate and keep the ones that have supplied appropriate credentials [Micheal2003].

The above example is one way of throttling resources, another way can be when managing system resources in POSIX, the method setrlimit() can be used to set limits for resources, and the method getrlimit() can be used to determine how many resources are available. In a condition where the resources are being used with a rapid rate, we can check when the current levels get close to the maximum using getrlimit(). We can then limit the allocation of further resources to privileged users; alternately, we can also begin to release resources for less-privileged users [CWE770]. This method would make sure that we free up resource and prevent a complete Denial of Service (DoS).

There are many options which a programmer can choose to implement throttling of resources. One method that a programmer can use to throttle resources is 1) stop giving out resources when you are short on them (give the resources to only few users) and 2) release the resources for specific users.

3. Conclusion

We have discussed two major countermeasures against CWE in the previous section. Although these two countermeasures are dealing with their own set of problems to solve our issue, they should be used in conjunction with each other. The first countermeasure mitigates the risk of encountering CWE-770 by eliminating the risk of causing resource exhaustion by the program's inadequacy. The second countermeasure is concerned with preventing resource exhaustion when the program is overwhelmed with requests and tasks. Since the two countermeasures solve two different problems associated with our weakness, it is a better approach to use them together.

References

[MITRE770] CWE- Individual Dictionary Definition: CWE-770: Allocation of Resources Without Limits or Throttling.
https://cwe.mitre.org/data/definitions/770.html

[CVE10908] CVE-2018-10908
https://www.cvedetails.com/cve/CVE-2018-10908/

[Wheeler2004] Wheeler, David A. ***Secure Programming for Linux and Unix HOWTO***.,3.6, 7.10,
https://dwheeler.com/secure-class/Secure-Programs-HOWTO.pdf

[CVE0213] CVE-2013-0213
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0231

[GITHUBCVE0213] CVE-2013-0231 : Log overflow
https://github.com/votd/vulnerability-of-the-day/wiki/Log-Overflow

[MITRECLASP2005] Secure Software Inc, ***The CLASP Application Security Process,*** 5.3.8
https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf

[BernsteinRlimit] Resource exhaustion, Managing UNIX resources, rlimit
http://cr.yp.to/docs/resources.html

[Micheal2003] Michael Howard, David LeBlanc. ***Writing secure code,*** Chapter 17, Resource Starvation

[LinuxPrlimit] prlimit(2) - Linux man page
https://linux.die.net/man/2/prlimit

[GforgLimit] Python | How to put limits on Memory and CPU Usage
https://www.geeksforgeeks.org/python-how-to-put-limits-on-memory-and-cpu-usage/

[LinuxWarnq] warnquota(8) - Linux man page
https://linux.die.net/man/8/warnquota

[LinuxQuotactl] quotactl(2) - Linux man page
https://linux.die.net/man/2/quotactl

[WheelerSlide3] Secure-Software-3-Buffer-Overflow.ppt, Do not create your own memory allocation system
https://dwheeler.com/secure-class/presentations/Secure-Software-3-Buffer-Overflow.ppt

[WheelerSlide8] Secure-Software-8-Errors.ppt, Error handling, Error code, Exception, Resource handling, Finally, Try-with-resources
https://dwheeler.com/secure-class/presentations/Secure-Software-8-Errors.ppt