Module 9 Memory corruption attacks and Security configurations

Introduction –

Most malicious attacks, viruses, and worms exploit low level programming errors to compromise the security of target systems.

Memory corruption occurs in a computer program when the contents of a memory location are unintentionally modified due to programming errors; this is termed violating memory safety. When the corrupted memory contents are used later in that program, it leads either to program crash or to strange and bizarre program behavior.

Memory corruption is one of the most intractable class of programming errors, for two reasons:

1. The source of the memory corruption and its manifestation may be far apart, making it hard to correlate the cause and the effect.
2. Symptoms appear under unusual conditions, making it hard to consistently reproduce the error.

Memory corruption errors can be broadly classified into four categories:

1. Using uninitialized memory: Contents of uninitialized memory are treated as garbage values. Using such values can lead to unpredictable program behavior.
2. Using none-owned memory: It is common to use pointers to access and modify memory. If such a pointer is a null pointer, dangling pointer (pointing to memory that has already been freed), or to a memory location outside of current stack or heap bounds, it is referring to memory that is not then possessed by the program. Using such pointers is a serious programming flaw. Accessing such memory usually causes operating system exceptions, that most commonly lead to a program crash (unless suitable memory protection software is being used).
3. Using memory beyond the memory that was allocated (buffer overflow): If an array is used in a loop, with incorrect terminating condition, memory beyond the array bounds may be accidentally manipulated. Buffer overflow is one of the most common programming flaws exploited by computer viruses, causing serious computer security issues (e.g. return-to-libc attack, stack-smashing protection) in widely used programs. In some cases programs can also incorrectly access the memory before the start of a buffer.
4. Faulty heap memory management: Memory leaks and freeing non-heap or un-allocated memory are the most frequent errors caused by faulty heap memory management.

Buffer Overflow attack –

A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code.

Buffer overflow is probably the best known form of software security vulnerability. Most software developers know what a buffer overflow vulnerability is, but buffer overflow attacks against both legacy and newly-developed applications are still quite common. Part of the problem is due to the wide variety of ways buffer overflows can occur, and part is due to the error-prone techniques often used to prevent them.

Buffer overflows are not easy to discover and even when one is discovered, it is generally extremely difficult to exploit. Nevertheless, attackers have managed to identify buffer overflows in a staggering array of products and components.

In a classic buffer overflow exploit, the attacker sends data to a program, which it stores in an undersized stack buffer. The result is that information on the call stack is overwritten, including the function's return pointer. The data sets the value of the return pointer so that when the function returns, it transfers control to malicious code contained in the attacker's data.

Although this type of stack buffer overflow is still common on some platforms and in some development communities, there are a variety of other types of buffer overflow, including Heap buffer overflow and Off-by-one Error among others. Another very similar class of flaws is known as Format string attack.

Buffer Overflow and Web Applications

Attackers use buffer overflows to corrupt the execution stack of a web application. By sending carefully crafted input to a web application, an attacker can cause the web application to execute arbitrary code – effectively taking over the machine.

Buffer overflow flaws can be present in both the web server or application server products that serve the static and dynamic aspects of the site, or the web application itself. Buffer overflows found in widely used server products are likely to become widely known and can pose a significant risk to users of these products. When web applications use libraries, such as a graphics library to generate images, they open themselves to potential buffer overflow attacks.

Buffer overflows can also be found in custom web application code, and may even be more likely given the lack of scrutiny that web applications typically go through. Buffer overflow flaws in custom web applications are less likely to be detected because there will normally be far fewer hackers trying to find and exploit such flaws in a specific application. If discovered in a custom application, the ability to exploit the flaw (other than to crash the application) is significantly reduced by the fact that the source code and detailed error messages for the application are normally not available to the hacker.

Consequences

- Category:Availability: Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
- Access control (instruction processing): Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program’s implicit security policy.
- Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe-string libraries and container abstractions could be introduced.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or mis¬use of mitigating technologies.

Environments Affected

Almost all known web servers, application servers, and web application environments are susceptible to buffer overflows, the notable exception being environments written in interpreted languages like Java or Python, which are immune to these attacks (except for overflows in the Interpretor itself).

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed, depending on environment.

How to Determine If You Are Vulnerable

For server products and libraries, keep up with the latest bug reports for the products you are using. For custom application software, all code that accepts input from users via the HTTP request must be reviewed to ensure that it can properly handle arbitrarily large input.

How to Protect Yourself

Keep up with the latest bug reports for your web and application server products and other products in your Internet infrastructure. Apply the latest patches to these products. Periodically scan your web site with one or more of the commonly available scanners that look for buffer overflow flaws in your server products and your custom web applications. For your custom application code, you need to review all code that accepts input from users via the HTTP request and ensure that it provides appropriate size checking on all such inputs. This should be done even for environments that are not susceptible to such attacks as overly large inputs that are uncaught may still cause denial of service or other operational problems.

Example –

The following sample code demonstrates a simple buffer overflow that is often caused by the first scenario in which the code relies on external data to control its behavior. The code uses the gets() function to read an arbitrary amount of data into a stack buffer. Because there is no way to limit the amount of data read by this function, the safety of the code depends on the user to always enter fewer than BUFSIZE characters.

```
...
        char buf[BUFSIZE];
        gets(buf);
        ...
```

The code in this example also relies on user input to control its behavior, but it adds a level of indirection with the use of the bounded memory copy function memcpy(). This function accepts a destination buffer, a source buffer, and the number of bytes to copy. The input buffer is filled by a bounded call to read(), but the user specifies the number of bytes that memcpy() copies.

```
...
        char buf[64], in[MAX_SIZE];
        printf("Enter buffer contents:\n");
        read(0, in, MAX_SIZE-1);
        printf("Bytes to copy:\n");
        scanf("%d", &bytes);
        memcpy(buf, in, bytes);

        ...
```

Heap Overflow –

A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as the POSIX malloc() call.

Consequences

- Availability: Buffer overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.
- Access control (memory and instruction processing): Buffer overflows often can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy.
- Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced.
- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All, although partial preventative measures may be deployed depending on environment.

Avoidance and mitigation

- Pre-design: Use a language or compiler that performs automatic bounds checking.
- Design: Use an abstraction library to abstract away risky APIs. Not a complete solution.

- Pre-design through Build: Canary style bounds checking, library changes which ensure the validity of chunk data, and other such fixes are possible, but should not be relied upon.
- Operational: Use OS-level preventative functionality. Not a complete solution.

Discussion

Heap overflows are usually just as dangerous as stack overflows. Besides important user data, heap overflows can be used to overwrite function pointers that may be living in memory, pointing it to the attacker's code.

Even in applications that do not explicitly use function pointers, the run-time will usually leave many in memory. For example, object methods in C++ are generally implemented using function pointers. Even in C programs, there is often a global offset table used by the underlying runtime.

Integer Overflow –

An integer overflow condition exists when an integer, which has not been properly sanity checked, is used in the determination of an offset or size for memory allocation, copying, concatenation, or similarly. If the integer in question is incremented past the maximum possible value, it may wrap to become a very small, or negative number, therefore providing a very incorrect value.

Consequences

- Availability: Integer overflows generally lead to undefined behavior and therefore crashes. In the case of overflows involving loop index variables, the likelihood of infinite loops is also high.
- Integrity: If the value in question is important to data (as opposed to flow), simple data corruption has occurred. Also, if the integer overflow has resulted in a buffer overflow condition, data corruption will most likely take place.
- Access control (instruction processing): Integer overflows can sometimes trigger buffer overflows which can be used to execute arbitrary code. This is usually outside the scope of a program's implicit security policy.

Exposure period

- Requirements specification: The choice could be made to use a language that is not susceptible to these issues.
- Design: Mitigating technologies such as safe string libraries and container abstractions could be introduced. (This will only prevent the transition from integer overflow to buffer overflow, and only in some cases.)

- Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.

Platform

- Languages: C, C++, Fortran, Assembly
- Operating platforms: All

Example

Integer overflows can be complicated and difficult to detect. The following example is an attempt to show how an integer overflow may lead to undefined looping behavior:

```
short int bytesRec = 0;

char buf[SOMEBIGNUM];


while(bytesRec < MAXGET) {

bytesRec += getFromInput(buf+bytesRec);

}
```

In the above case, it is entirely possible that bytesRec may overflow, continuously creating a lower number than MAXGET and also overwriting the first MAXGET-1 bytes of buf.

Race Condition –

A race condition occurs when a pair of routine programming calls in an application do not perform in the sequential manner that was intended per business rules. It is a timing event within software that can become a security vulnerability if the calls are not performed in the correct order.

1. Race condition in checking for certificate revocation

If the revocation status of a certificate is not checked before each privilege requiring action, the system may be subject to a race condition, in which their certificate may be used before it is checked for revocation.

Consequences

- Authentication: Trust may be assigned to an entity who is not who it claims to be.
- Integrity: Data from an untrusted (and possibly malicious) source may be integrated.

- Confidentiality: Date may be disclosed to an entity impersonating a trusted entity, resulting in information disclosure.

Exposure period

- Design: Checks for certificate revocation should be included in the design of a system
- Design: One can choose to use a language which abstracts out this part of the authentication process.

Platform

- Languages: Languages which do not abstract out this part of the process.
- Operating platforms: All

2. Race condition in signal handler

Race conditions occur frequently in signal handlers, since they are asynchronous actions. These race conditions may have any number of Problem Types and symptoms.

Consequences

- Authorization: It may be possible to execute arbitrary code through the use of a write-what-where condition.
- Integrity: Signal race conditions often result in data corruption.

Exposure period

- Requirements specification: A language might be chosen which is not subject to this flaw.
- Design: Signal handlers with complicated functionality may result in this issue.
- Implementation: The use of any non-reentrant functionality or global variables in a signal handler might result in this race conditions.

Platform

- Languages: C, C++, Assembly
- Operating platforms: All

3. Race condition in switch

If the variable which is switched on is changed while the switch statement is still in progress, undefined activity may occur.

Consequences

- Undefined: This flaw will result in the system state going out of sync.

Exposure period

- Implementation: Variable locking is the purview of implementers.

Platform

- Languages: All that allow for multi-threaded activity
- Operating platforms: All

4. Race condition within a thread

If two threads of execution use a resource simultaneously, there exists the possibility that resources may be used while invalid, in turn making the state of execution undefined.

Consequences

- Integrity: The main problem is that if a lock is overcome data could be altered in a bad state.

Exposure period

- Design: Use a language which provides facilities to easily use threads safely.

Platform

- Languages: Any language with threads
- Operating platforms: All

5. Time of check, time of use race condition

Time-of-check, time-of-use race conditions occur when between the time in which a given resource is checked, and the time that resource is used, a change occurs in the resource to invalidate the results of the check.

Consequences

- Access control: The attacker can gain access to otherwise unauthorized resources.
- Authorization: race conditions such as this kind may be employed to gain read or write access to resources which are not normally readable or writable by the user in question.

- Integrity: The resource in question, or other resources (through the corrupted one), may be changed in undesirable ways by a malicious user.
- Accountability: If a file or other resource is written in this method, as opposed to in a valid way, logging of the activity may not occur.
- Non-repudiation: In some cases it may be possible to delete files a malicious user might not otherwise have access to, such as log files.

Exposure period

- Design: Strong locking methods may be designed to protect against this flaw.
- Implementation: Use of system APIs may prevent check, use race conditions.

Platform

- Languages: Any
- Platforms: All

Controls

- Design: Ensure that some environmental locking mechanism can be used to protect resources effectively.
- Implementation: Ensure that locking occurs before the check, as opposed to afterwards, such that the resource, as checked, is the same as it is when in use.