RECOGNIZING AND PREVENTING TIME-OF-CHECK TO TIME-OF-USE VULNERABILITIES

Christopher Hacking – chacking[at]isecpartners[dot]com

iSEC Partners, part of NCC Group
123 Mission Street, Suite 900
San Francisco, CA 94105
https://www.isecpartners.com

March 2, 2015

Abstract

Time-Of-Check to Time-Of-Use (TOCTOU) vulnerabilities may occur where a privileged component attempts to validate untrusted input, and result in elevation of privileges. This paper describes the characteristics of TOCTOU to enable developers to recognize this class of vulnerabilities, and proposes solutions for various scenarios in which TOCTOU is likely to be found. A selection of examples demonstrates the problem and several solutions.

1 INTRODUCTION

Time-of-Check to Time-of-Use, often abbreviated as "TOCTOU" or "TOCTTOU", describes a class of security vulnerabilities that occur when a privileged system component validates (checks) an attacker-controllable resource prior to consuming (using) it, and an untrusted attacker is able to modify the resource between the check and the use. TOCTOU vulnerabilities typically lead to elevation of privilege exploits, where a malicious input will cause unintended behavior in the privileged component and the attacker can modify that input to make it malicious *after* it passes validation.

This paper is aimed at software architects, developers, and testers. Our goal is to help avoid situations in which TOCTOU vulnerabilities might occur, recognize them when they are present, and remedy them. There are too many variants of TOCTOU to discuss each one individually, so we will instead attempt to describe classes of scenarios in which TOCTOU vulnerabilities occur, and provide representative examples. We hope that this will enable developers to recognize similar patterns in their own code.

Preventing TOCTOU vulnerabilities is an area which has received considerable research. However, most of this research has either focused on a specific form of TOCTOU vulnerability, or the proposed solutions have been found inadequate. Additionally, runtime testing for TOCTOU is difficult, as each potential vulnerability may require a custom test and TOCTOU vulnerabilities are usually non-deterministic. Therefore, TOCTOU issues are typically identified during design or code review. This paper will not discuss automated tests to recognize TOCTOU vulnerabilities.

2 Description of TOCTOU Vulnerabilities

TOCTOU vulnerabilities have been known since at least the 1970s. However, they occur in many forms, and new examples are still being discovered today. They can occur on all platforms and all languages. The attacker usually needs to be local, but in some cases can be remote. The typical impact is elevation of privileges.

2.1 The problem

Systems frequently contain trust boundaries, which divide a system between trusted components (which are privileged) and untrusted components and actors (which are not). Examples of trust boundaries are the user-kernel boundary, the user account boundary, and the machine boundary. In most such systems, a privileged component will sometimes need to trust (use) input which comes across a trust boundary. To ensure that this input is trustworthy despite its source, the privileged component will validate (check) the input before use.

TOCTOU vulnerabilities occur when the attacker can modify the resources *after the check but before the use*. The privileged component now trusts the input, but the input is *no longer* trustworthy. If the behavior of the privileged component is determined by the input, then the unprivileged attacker now has elevated privileges.

2.2 Race conditions

The most common TOCTOU vulnerabilities are race conditions, where an operation which is assumed to be atomic (such as checking a file path or a pointer, and then using it) may be interrupted on a multitasking operating system, allowing the attacker's code to execute and change the resource between check and use. After a resource is checked, the vulnerable code is "racing" with the attacker to use the resource before it gets changed.

Exploiting a race condition is typically non-deterministic, but in many cases the attacker can repeat the attack until it succeeds, and will only need to succeed once. A well-known historical TOCTOU race condition in the BSD 4.3 `mail` utility could usually be exploited in about a minute.

There have been a number of proposed mitigations for race conditions. For example, if you require that the race be won many times in a row, naïve exploitation becomes nearly impossible. [1] However, more advanced attacks may still occur. [2] Our recommended solutions are deterministic and avoid the race entirely.

3 Recognizing TOCTOU

3.1 Characteristics of TOCTOU

All TOCTOU vulnerabilities have certain elements in common.

- A **trusted component** that has higher privileges than the attacker.
- An **untrusted resource** (usually an input) which the attacker can control.
- An **undesirable action** which the trusted component will take if it trusts (uses) a malicious input.
- A **check** by which the trusted component determines whether it can trust the input.
- An **opportunity** for the attacker to modify the resource between the check and the use.

The trusted component must have privileges that a potential attacker lacks, or there is no threat. It does not matter if the component is not fully trusted by the system, however. For example, the programs that User A runs are typically trusted to read and write User A's files, while programs that User B runs are not. Regardless of whether or not the users have administrative capabilities, most modern operating systems have a trust boundary between user accounts. It is important to not allow one user to access another user's data.

The untrusted resource or input may be supplied by the attacker or chosen by the trusted component, but it must be attacker-controllable. A file path where the attacker has no write access is not at risk of a TOCTOU vulnerability, but a file where the attacker can control even one directory in the path to that file may be a risk.

The way in which the trusted component uses the resource must be of potential use to the attacker. A script that will execute in the trusted context is of interest to the attacker; an image used to represent the attacker's user account is much less sensitive.

The distinguishing characteristic of the check in a TOCTOU-vulnerable system is that an input which passes the check is considered trusted. If the component declines to ever trust the input – for example, by dropping its own privileges to those of the input's source while the input is being used – then it is not vulnerable to TOCTOU (unless the result of checking the input's source can be attacker-controlled).

The core of a TOCTOU vulnerability, however, is the opportunity for an attacker to modify the resource after it has been checked, but before it is used. When the attacker can invalidate the check's result but the trusted component still trusts the input, the result is equivalent to just removing the check and trusting any input.

3.2 Multiple retrievals

Typically, TOCTOU vulnerabilities occur when a resource is indicated by a path, name, or other reference or identifier, and is checked and used via multiple retrievals of the identified item. For example, a file which is checked via its name, then opened by name (and used in some manner) is potentially vulnerable; file systems can be manipulated such that a given path and name which refers to one file during the check may refer to another during use. In addition to file paths, this can apply to network addresses, pointer variables in multithreaded programs, and more.

The key is the multiple retrievals or dereferences. If the resource is loaded in such a way that an external attacker cannot modify it, and the loaded copy is used for both the check and the use without a new load, then a TOCTOU vulnerability cannot exist. However, be aware that the retrieved resource often includes its environment. For example, if checking whether a named file exists, the resource being retrieved is the state of that portion of the file system (including metadata such as permissions), not just a particular file name.

Some TOCTOU vulnerabilities occur when the attacker can control the channel over which the resource is loaded. If software erroneously trusts a communication mechanism to be immune to tampering, it might request data over the channel once to check it for validity, and then again (assuming it to be unchanged) for usage. The response to the second request could be tampered with by the attacker, leading to the use of unchecked data. The most common place for such a problem to arise would be in the transfer of large resources between machines; if the attacker controls the other machine or can intercept and modify network traffic, it is easy to present different inputs for the check and the use. Such a TOCTOU vulnerability is often not a race condition; the attacker can simply switch the resource after the first retrieval is completed.

4 Remediating TOCTOU

There are several general approaches to preventing TOCTOU vulnerabilities. If the check and the use can be made atomic, there will be no opportunity for an attacker to interfere. Alternatively, perform the check and the use out of the attacker's reach. Other solutions may be possible, and in some situations may be required.

However, the simplest solution that will work is usually the best choice, as it reduces the risk of introducing a new vulnerability. Also, some of these proposed fixes are usually platform-specific, so developers of portable code might need to avoid them. Ensure that your chosen solution is safe on all supported platforms.

4.1 ATOMIC CHECK AND USE

Sometimes it is possible to perform the desired check as part of the use. If both actions are atomic from the attacker's perspective, then there is no TOCTOU vulnerability. APIs (or optional parameters to the APIs you are currently using) may incorporate the check and the use into a single operation, such that the attempted use fails if the check would have failed. In such cases, a separate step to check may be redundant from a security perspective (existing only to, for example, avoid the performance cost of failed attempt at use).

4.2 CACHING THE RETRIEVED RESOURCE

In many cases, the solution to TOCTOU is to avoid the repeated retrieval of the resource. By retrieving the resource only once and keeping it internally (where the attacker cannot modify it), the check and use become effectively atomic from the attacker's perspective regardless of how much time actually passes between check and use. This approach is most effective when the resource in question is easy to store, such as a reference to a file; rather than using a file path for each operation (and risking the attacker changing elements of the path), open the file once and store the file descriptor.

4.3 ENSURE THE RESOURCE IS UNCHANGED

Sometimes avoiding repeated retrievals of the resource is impractical, such as when a very large amount of data would need to be cached. In such cases, a hybrid approach may be used: on the initial retrieval to check the resource, also create and store a cryptographic hash of the data, then verify the hash on the subsequently-retrieved data prior to using it.

In some cases it is impractical to store the entire resource in working memory. If the resource is retrieved in chunks of known size and offset, each chunk can be hashed during the check, and the hashes stored. During use, re-hash each chunk and verify that this hash matches the cached value before using the chunk. While this will impose a performance impact, it will avoid the risk of using untrustworthy resources.

4.4 PREVENT MODIFICATION OF THE RESOURCE

In some scenarios, it is possible to "lock" a resource, preventing modification by an attacker, while the trusted component performs the check and use. However, it is vital to distinguish between locking the resource and locking its reference. For example, if the resource is referenced by path, it is vital to lock the entire path rather than just locking the final portion where the resource is named.

4.5 USE AN UNPRIVILEGED PROCESS

As mentioned in section 3.1, the first characteristic of a TOCTOU vulnerability is the presence of a trusted component with privileges greater than the attacker currently possesses. Sometimes elevated privileges are not actually required. In that case, the component should use the resource in a process or thread with the same privileges as the potential attacker. Under this remediation, the check is generally no longer required at all; the input is always treated as untrusted.

5 Examples of TOCTOU Vulnerabilities

The following examples are intended to illustrate a diverse sampling of TOCTOU vulnerabilities. This is not a comprehensive list of either vulnerable mechanisms or scenarios where such mechanisms may be used.

5.1 Program privilege checks

5.1.1 Description

This section could have just been called "the POSIX `access` system call", but while most people now know not to use `access`[1], it has been re-implemented just as insecurely under other names and on other platforms. This is possibly *the* classic example of a TOCTOU vulnerability. The function takes a file path as a string and an access mask as an integer, and checks whether the program's real user ID – that is, the ID of the user who launched the program, even if the executable has the setuid bit set – has the specified access to that file. Its purpose is to allow a high-privilege program to take actions on a user's behalf, but only if the user would be able to take that action already.

Part of the reason that `access` is such a popular example of TOCTOU is because it is inherently vulnerable. This is because the file to check is referenced by path, and there is neither any mechanism for preventing the file referenced by that path from changing nor any way to verify that the file checked by `access` is the same one being used later. Any function which uses file paths for a similar purpose is probably vulnerable.

Note that this does not just apply to file system resources. For example, on Microsoft Windows, an issue of this type could involve a registry key, named pipe, driver object, or any other securable object.

5.1.2 Code

Simple POSIX C example showing abuse of a setuid program to overwrite the system authentication file:

Vulnerable system code

```
// RootProg (runs setuid root)
// "Safely" open a file for writing
int openUserFile(char *file) {
    int fd;
    // Check real UID access
    if (access(file, W_OK)) {
       return -1;
    } // Else, safe to write!
    // Note: check was above, use is below!
    fd = open(file, O_WRONLY);
    return fd;
}
```

Attacker code

```
// AttackProg (runs as limited user)
while (!isFileOverwritten()) {
    system("touch ./safefile");
    // Launch the vulnerable program in the bg
    system("RootProg ./safefile &");
    // Change the target file into a link to
    // a file that we can't write normally
    remove("./safefile");
    link("/etc/shadow", "./safefile");
    usleep(10);
    remove("./safefile");
}
```

5.1.3 Remediation

One of the simplest ways to fix this vulnerability is to make the check and the use a single, atomic operation (from the perspective of the file system). In the case of `access`, the OS already checks the file permissions

[1] The Linux manual specifically notes the security vulnerability and recommends against using this system call: http://linux.die.net/man/2/access

when opening a file. To make it check against the real ID, it is sufficient to temporarily drop the effective ID to match the real ID:

Patched system code

```
// RootProg (runs setuid root)
// Safely open a file for writing
int openUserFile(char *file) {
    int fd = -1;
    // Drop effective UID root for real UID
    if (!seteuid(getuid())) {
        // Now open file, let OS check perms
        fd = open(file, O_WRONLY);
        // Restore root permissions
        seteuid(0);
    }
    return fd;
}
```

Similar solutions are available on all modern platforms, although the exact implementation may vary. On the Microsoft Windows platform, for example, a thread on the privileged process may impersonate[2] the caller.

Another option is to simply avoid having the privileged process perform the task; sometimes it is possible to have the unprivileged process perform the operation directly. However, if the unprivileged process needs some information from the privileged one before the operation can be carried out, it is important to secure the channel used to communicate between the processes to avoid introducing a new vulnerability.

Other options exist, such as opening the file descriptor and then using `fstat` to confirm access for the real ID (essentially re-implementing `access` with a file descriptor in place of a path) works. However, TOCTOU vulnerabilities (as with other classes of software bug) should generally be fixed in the simplest manner possible; more complex fixes risk either just moving the vulnerability, or introducing a new one.

5.2 TEMPORARY FILE CREATION

5.2.1 DESCRIPTION

Programs frequently create temporary files to cache data too big for memory, to pass data between processes, or various other reasons. These files might contain sensitive data that untrusted processes should not have access to. Alternatively, privileged processes may create the files and write data into them that an attacker would like to have written into a different file.

5.2.2 GETTEMPFILENAME

The Win32 API provides a function for creating temporary files, `GetTempFileName`[3]. Despite the function's name, it will actually create the file and then close it (providing the name to the function's caller). Because no particular access controls are applied to the created file, it will simply inherit the security of the directory in which it resides. If the file is created in a limited-user-writable location, limited users can delete the created file and replace it.

[2] Impersonation: http://msdn.microsoft.com/en-us/library/windows/desktop/aa376391(v=vs.85).aspx
[3] `GetTempFileName`: http://msdn.microsoft.com/en-us/library/windows/desktop/aa364991(v=vs.85).aspx

This is a TOCTOU issue because the API checks for an unused file name, but the file is created at the time of the check and then closed. Securing or using the file will require re-opening the file by its path, allowing an attacker to replace the file (or a folder in the path) first. This is a common problem any time a program needs to create a new file in a shared location.

5.2.3 OTHER TEMPORARY FILE FUNCTIONS

POSIX has its own insecure temporary file functions, such as `mktemp`, `tmpnam`, and `tempnam`. These functions do not create the temp file, and instead merely return a file name that did not exist at the time of the check.

The C standard function `tmpfile` creates a unique file and returns an open `FILE*` pointer which can be safely used, but offers no control over the file's location or creation mode.

The POSIX function `mkstemp` (and its siblings in glibc, such as `mkostemps`) create a temp file securely and return a usable file descriptor, but they also return the full path of the file. This path must not be trusted to refer to the same file if used in another `open` call (or similar API).

5.2.4 CODE

Example Win32 C++ and C# code showing abuse of the `GetTempFileName` API with a privileged system component that inadvertently overwrites another system component. This particular example requires that the attacker have the ability to create symbolic links, which on Windows is not granted by default.

Vulnerable system code

```
// InstallSvc (runs as TrustedInstaller)
// Store network-retrieved data in a temp file
HANDLE storeTempData(char *data, DWORD len) {
    // Get a new, unique temp file name
    WCHAR p[MAX_PATH];
    GetTempFileNameW(L"\\TMP", L"Ins", 0, p);
    // Now that we checked for a file nobody
    // else is using, we can use it ourselves
    HANDLE h = CreateFile(
p, GENERIC_ALL, 0, NULL, OPEN_EXISTING,
FILE_FLAG_DELETE_ON_CLOSE, NULL);
    if (!h || INVALID_HANDLE_VALUE == h) {
        // File is open or doesn't exist
        return NULL;
    }
    // Restrict access and write to the file
    LockFile(h, 0, 0, len, 0);
    if (WriteFile(h, data, len, &len, NULL))
        return h;
    // Something went wrong if we get here
    CloseHandle(h);
    return NULL;
}
```

Attacker code

```
// AttackProg (limited user + symlinks)
// Overwrites UAC prompter (SYSTEM process)
void Elevate () {
    // Monitor the temp directory
    FileSystemWatcher fsw = new
        FileSystemWatcher("\\TMP", "Ins*");
    fsw.Created += (ob, ev) => {
        // Event handler to race InstallSvc
        File.Delete(ev.FullPath);
        if (CreateSymbolicLink(ev.FullPath,
            @"\Windows\System32\consent.exe",
            0)) { // Wrapper around Win32 API
            // Wait for write then trigger UAC
            Thread.Sleep(1000);
            Process.Start("devmgmt.msc");
        }
    };
    fsw.EnableRaisingEvents = true;
    while (!IsFileOverwritten()) {
        // Trigger InstallSvc over network
        SendPayloadToElevate();
    }
}
```

5.2.5 REMEDIATION

There are multiple approaches to fixing temporary file TOCTOU. One of the best is to use a temporary file function which returns a still-open file descriptor or HANDLE, making the operation atomic and eliminating the opportunity for an attacker to interfere with the file. Unfortunately, while POSIX has the `mkstemp` APIs, Win32 lacks any such functions. The C standard function `tmpfile` can be used on most platforms, but offers no control over where the file is created (notably, the Microsoft C runtime creates the file in the root of the drive and may fail due to insufficient permissions[4]).

Another option is to create the temporary file directly, requiring that it not yet exist. This combines the check (for file existence) and the use (opening a new, temporary file) in a single, atomic operation; if the specified file exists, the function will fail. On POSIX, including the `O_CREAT|O_EXCL` flags to `open` will require that the file not yet exist. On Windows, use the `CREATE_NEW` option in the `dwCreationDisposition` parameter of the `CreateFile` API (you should also set `dwShareMode` to 0 and/or specify a very restrictive security descriptor). No such option exists universally for C library functions such as `fopen`, although some implementations (such as glibc) support the "x" flag to require this behavior.[5]

An additional option usable in some scenarios is to "lock" a file or directory such that no other process can modify it. On Windows, this can be done by opening a HANDLE to any file or directory without any share flags; the opened item and every element of its path will be immutable to other processes until the HANDLE is closed (or the process exits). If the path specified in `CreateFile` contains one or more reparse points[6] (such as directory junctions, which can be created without any special privileges) then the path that gets locked will be the resolved path; the reparse points themselves will not be locked. To safely use the locked path, retrieve the path of the locking HANDLE[7] instead of using the path by which the HANDLE was opened.

Finally, one can use a location that the attacker cannot access. However, it is important to remember that the entire path must be safe; if the attacker can replace any directory in the path with a symlink or junction to a location that the attacker can write to, then TOCTOU attacks are still possible.

5.3 MEMORY LOCATIONS

5.3.1 DESCRIPTION

When memory is shared between functions executing at different privilege levels, there is a threat of the less-privileged process attempting to use the shared memory to gain elevated privileges. If the trusted component checks a value supplied via the shared memory and then attempts to use it for something sensitive, the less-privileged code might modify the shared value to invalidate the check result before the value is used. This can occur between processes, or within a process when threads are executing at different privilege levels (such as user-mode and kernel-mode).

5.3.2 CODE

Example of code one might find in an NT kernel-mode driver's handler for the `DeviceIoControl` function (an IOCTL handler). In this example, the driver uses neither buffered nor direct I/O[8], so it operates in the address space of the caller. As kernel-mode code, however, it is able to read and write kernel-mode addresses. Some variable assignments and error handling are omitted for brevity.

[4] `tmpfile` on MSDN: http://msdn.microsoft.com/en-us/library/x8x7sakw.aspx
[5] http://linux.die.net/man/3/fopen - see the section "Glibc notes"
[6] Reparse points: http://msdn.microsoft.com/en-us/library/windows/desktop/aa365505(v=vs.85).aspx
[7] Path from HANDLE: http://msdn.microsoft.com/en-us/library/windows/desktop/aa366789(v=vs.85).aspx
[8] Non-buffered I/O: http://msdn.microsoft.com/en-us/library/windows/hardware/ff565432(v=vs.85).aspx

Vulnerable driver code

```
// Executes in caller's address space (neither buffered nor direct I/O)
// Writes to a user-specified address specified in a structure in the output buffer
try {
    UserData *pData = (UserData*)(Irp->UserBuffer);
    // Make sure the destination buffer is large enough or raise an exception
    if (pData->buflen < outputLen) ExRaiseStatus(STATUS_INVALID_PARAMETER);
    // Ensure the destination buffer is user-mode and is mapped or raise an exception
    ProbeForWrite(pData->buffer, outputLen, 1);
    // If we got here, the probe succeeded and we can write to the user-mode buffer
    RtlCopyMemory(pData->buffer, outputBuffer, outputLen);
} except (EXCEPTION_EXECUTE_HANDLER) {
    // Handle exceptions gracefully…
}
```

In this example, if the address stored at `pData->buffer` is changed after the `ProbeForWrite` call but before the `RtlCopyMemory` (`memcpy`) call, the safety guarantees of the `ProbeForWrite` check will no longer be valid. The `UserData` struct is in user-mode memory; any thread executing in parallel with the one that called the IOCTL could modify that address to do something like point at a kernel memory address, allowing untrusted user-mode code to overwrite arbitrary kernel data.

5.3.3 REMEDIATION

In this case, making the check and use atomic is probably possible, but at great cost and complexity; all non-caller threads in the calling process would need to be suspended, as would every other untrusted process that could debug the caller. Instead, copy the sensitive values to a location where an attacker cannot modify them and then use those cached values for the probe and the write.

Patched driver code

```
// Executes in caller's address space (neither buffered nor direct I/O)
// Writes to a user-specified address specified in a structure in the output buffer
try {
    UserData *pData = (UserData*)(Irp->UserBuffer);
    // Make sure the destination buffer is large enough or raise an exception
    if (pData->buflen < outputLen) ExRaiseStatus(STATUS_INVALID_PARAMETER);
    // Copy the address locally
    void *buffer = pData->buffer;
    // Ensure the destination buffer is user-mode and is mapped or raise an exception
    ProbeForWrite(buffer, outputLen, 1);
    // If we got here, the probe succeeded and we can write to the user-mode buffer
    RtlCopyMemory(buffer, outputBuffer, outputLen);
} except (EXCEPTION_EXECUTE_HANDLER) {
    // Handle exceptions gracefully…
}
```

An attacker could still modify memory between the probe and the write. For example, if the user buffer gets unmapped before the copy operation, the copy will fail. However, the whole operation is in a `try` block, and the attacker cannot change the address to which the write is attempted after probing. For most drivers, using a safer method of passing data (such as Buffered I/O[9], where the kernel safely copies data back to user-mode) is also an option. Regardless, for nearly all cases of memory TOCTOU, the simplest solution is to cache the input out of the attacker's reach and perform both the test and use on the cached data.

[9] http://msdn.microsoft.com/en-us/library/windows/hardware/ff565356(v=vs.85).aspx

5.4 Network Resources

5.4.1 Description

This is an interesting case, because it is frequently not a race condition. TOCTOU over the network usually occurs when a resource (could be a file, a media stream, or something else entirely) is downloaded and the client checks some characteristic of it but does not store the entire download. If the client then wants to use the resource (because the check succeeded) the download must be repeated. In the case of an insufficiently secure connection, the attacker can intercept and modify the resource when it is retrieved (downloaded) the second time. In the case of an untrustworthy server, the attacker can modify the resource for the second load. Since a network attacker or server can easily control the flow of data, the attacker has no need to “race” to modify the resource between subsequent retrievals and thus, this attack can succeed 100% of the time.

5.4.2 Code

Example of a C# utility that retrieves an update package, checks to make sure it was signed by the vendor, and if so, installs it. The update utility has the public key for the signing authority, and the file’s signature is located at the head of the file. The file may be too big for the utility to store locally, so it will be downloaded and its hash computed in blocks. If the signature verifies for the hashed data, the update will be installed.

Vulnerable update retrieval

```
// Gets the update package’s signature, then builds a hash of the data to verify.
// returns true if the update verifies, in which case it will be re-downloaded and installed
bool CheckUpdateSignature (NetworkStream update, // open connection to the server
                           AsymmetricSignatureDeformatter verif, // keyed signature verifier
                           HashAlgorithm hash) { // cryptographic hash function used by sig
    byte[] sig = new byte[SIGNATURE_SIZE], len = new byte[8], buf = new byte[1 << 20];
    long remaining; // Size of the download that we have not yet hashed
    int bytes;
    try { // Get the signature; it’s at the head of the network stream
        if (update.Read(sig, 0, SIGNATURE_SIZE) < SIGNATURE_SIZE) return false;
        // Get the length (next 8 bytes, network byte order)
        if (reader.Read(len, 0, 8) < 8) return false;
        if (BitConverter.IsLittleEndian) Array.Reverse(len);
        remaining = BitConverter.ToInt64(len);
        while (remaining > 0) { // Get the update package 1MB at a time
            bytes = update.Read(buf, 0, (int)(Math.Min(remaining, (1L << 20))));
            remaining -= bytes;
            if (remaining > 0)
                hash.TransformBlock(buf, 0, bytes, buf, 0); // Update hash with this block
            else hash.TransformFinalBlock(buf, 0, bytes);
        }  // We now have a hash of the while update package
        return verif.VerifySignature(hash.Hash, sig);
    } catch (Exception ex) {
        // Handle exceptions gracefully…
}   }
```

If the attacker controls the server (perhaps the update utility was tricked to connect to the wrong server, or the attacker compromised the server), or controls the network traffic, then the attacker controls what data the updater sees on each download. When this check is performed, the attacker sends a valid update package and its signature. However, when the updater goes to re-download the update package for use, the attacker can substitute malicious code that is not signed.

5.4.3 Remediation

There are a few options here. The first is to download the data only once; store it locally and check the local copy, then use that local copy if the check passes. If the data might not fit in RAM, a temporary file may be an option (make sure to avoid the pitfalls in section 5.2). If a temporary file will not work – for example, if the device doesn't have enough spare storage to hold the full download – then each chunk of the download will need to be verified individually.

The best way to do that, assuming the ability to verify the data as a whole, is to cryptographically hash each individual chunk and store the digests (hopefully there is storage for this). Then, when re-downloading the data for use, verify the hash of each chunk – while it is held in memory – against the stored value, before the data gets used.

6 Conclusion

TOCTOU vulnerabilities come in a variety of forms, and can appear in any language and on any platform. It is not possible to provide an exhaustive exploration of the topic any more than it would be possible to provide a list of all the types of checks a program might make on a resource obtained across a trust boundary. However, the descriptions and examples in this paper will hopefully enable software architects, developers, and testers to recognize TOCTOU risks in their designs and code, and to remedy them.

This paper has described the characteristics of a TOCTOU vulnerability: the trusted component that takes a reference to a resource or input from an untrusted source, retrieves the input and checks its safety, and uses the now-trusted resource in a way that will have a negative effect if the resource is not trustworthy. We have identified the key aspect of the vulnerability: the multiple retrievals (for check, then use) with an opportunity for the attacker to modify the resource between check and use. We have provided broad categories of remediation for TOCTOU: using an atomic check-and-use, caching the resource where the attacker cannot modify it before use, ensuring the resource is unchanged between check and use, and dropping privileges to match those of the attacker. For several common forms of TOCTOU vulnerability, we have provided examples and code to help developers recognize such vulnerabilities in their own systems, and have provided potential solutions to each example.

References

[1] D. Dean and A. J. Hu, "Fixing Races for Fun and Profit: How to use access(2)," in *13th USENIX Security Symposium*, San Diego, CA, 2004.

[2] N. Borisov, R. Johnson, N. Sastry and D. Wagner, "Fixing races for fun and profit: how to abuse atime," in *14th USENIX Security Symposium*, Baltimore, MD, 2005.