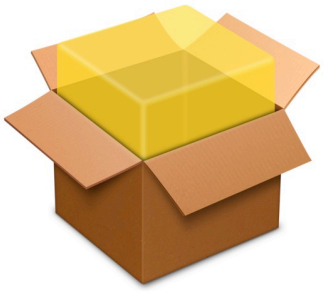
SECTOR

powered by Computest

January 13, 2023

Bad things come in large

packages: .pkg signature verification bypass on macOS



macOSPublicBetaAccessUtility.pkg
Installer package - 4,29 GB

Code signing of applications is an essential element of macOS security. Besides signing applications, it is also possible to sign installer packages (.pkg files). During a short review of the xar source code, we found a vulnerability

(CVE-2022-42841) that could be used to modify a signed installer package without invalidating its signature. This vulnerability could be abused to bypass Gatekeeper, SIP and under certain conditions elevate privileges to root.

Background

Installer packages are based on xar files with a number of predefined file names. The method for signing installer packages is the same as generating signed xar files, so to start we'll explain how that file format works.

A xar file consists of 3 parts: a fixed length header, a table of contents (TOC) and what is called the "heap".

The header contains a number of fields, including the hashing algorithm that is used throughout the file (typically still SHA1) and the size of the TOC.

The TOC is a zlib-compressed XML document. This document lists for each file included in the archive the start address and length where the contents can be found on the heap, starting with 0 for the first byte directly after the TOC. Each file in the archive can be compressed independently by specifying an encoding, so when creating an archive file it is possible to

choose the optimal way of storing each file.

For all files, a hash is included in the TOC of both the uncompressed and compressed data, using the hashing algorithm specified in the header.

For example:

```
<file id="4">
  <data>
    <length>430</length>
    <offset>2533</offset>
    <size>654</size>
    <encoding style="application/x
    <extracted-checksum style="sha
    <archived-checksum style="sha1
  </data>
  <FinderCreateTime>
    <nanoseconds>0</nanoseconds>
    <time>1970-01-01T00:00:00</tim
  </FinderCreateTime>
  <ctime>2022-10-03T17:54:54Z</cti
  <mtime>2022-10-03T17:54:54Z</mti
  <atime>2022-10-03T17:54:54Z</ati
  <group>wheel</group>
  <gid>0</gid>
  <user>root</user>
  <uid>0</uid>
  <mode>0644</mode>
  <deviceno>16777241</deviceno>
```

```
    <inode>33</inode>
    <type>file</type>
    <name>PackageInfo</name>
  </file>
```

Even xar files that are not signed have these hashes and so the integrity can be verified when extracting a file.

To verify the integrity of the entire archive, the TOC also lists the location on the heap where a value known as the “TOC hash” is stored. In practice this is usually at offset 0:

```
  <checksum style="sha1">
    <offset>0</offset>
    <size>20</size>
  </checksum>
```

The value stored here must be equal to the hash of the compressed TOC data and this is verified when the archive is opened. The reason this is included on the heap and not in the TOC itself is that this would create a cyclic dependency: adding this value into the TOC would change the TOC and the TOC hash again.

This hash indirectly guarantees the integrity of all files in the archive: for each file, the

`extracted-checksum` in the TOC ensures the integrity of that file. The integrity of the TOC is covered by the TOC hash. This construction has the nice benefit that a single file can be extracted and validated without having to validate the entire archive. This means it is possible to extract files from xar archives without completely reading the archive, or possibly even without completely downloading it.

Signed xar files additionally contain a signature element with a certificate chain in the TOC:

```
<signature style="RSA">
  <offset>20</offset>
  <size>256</size>
  <KeyInfo xmlns="http://www.w3.or
    <X509Data>
      <X509Certificate>MIIFdjCCBF6
      <X509Certificate>MIIEbDCCA1S
      <X509Certificate>MIIEuzCCA6O
    </X509Data>
  </KeyInfo>
</signature>
```

The signature itself is also stored on the heap (for the same cyclic dependency reason). The data used for generating the signature is the TOC hash. This signature therefore ensures

the authenticity of all files in the archive.

Interestingly, this design does mean that data on the heap that is not included in any of the ranges can be modified without invalidating the signature. For example, appending more data to a xar file will always keep the TOC hash and signature valid.

The vulnerability

For signed packages, the TOC hash needs to be used for two different checks:

> The computed TOC hash needs to be equal to the TOC hash stored on the heap.
> The signature and the certificates need to correspond to the TOC hash.

This is implemented in the following locations in the xar source code.

Here, the computed TOC is compared to the value stored on the heap.

https://github.com/apple-oss-distributions/xar/blob/f67a3a8c43fdd35021fd3d1562b62d2da32b4f4
L484

```
/* if TOC specifies a location for
 * we read the checksum from there
 * with a signature, because the s
 * the checksum at the specified l
 */
const char *value;
uint64_t offset = 0;
uint64_t length = 0;
if( xar_prop_get( XAR_FILE(ret) ,

    if (value) {
        errno = 0;
        offset = strtoull( value,
        if( errno != 0 ) {
            fprintf(stderr, "check
            xar_close(ret);
            return NULL;
        }
    } else {
        fprintf(stderr, "checksum/
        xar_close(ret);
        return NULL;
    }
}

[...]

XAR(ret)->heap_offset = xar_get_he
if( lseek(XAR(ret)->fd, XAR(ret)->
    xar_close(ret);
    return NULL;
}
```

```
[...]

size_t tlen = 0;
void *toccksum = xar_hash_finish(X
XAR(ret)->toc_hash_ctx = NULL;

if( length != tlen ) {
    free(toccksum);
    xar_close(ret);
    return NULL;
}

// Store our toc hash upon archive
// has changed or been tampered wi
XAR(ret)->toc_hash = malloc(tlen);
memcpy(XAR(ret)->toc_hash, toccksu
XAR(ret)->toc_hash_size = tlen;

void *cval = calloc(1, tlen);
if( ! cval ) {
    free(toccksum);
    xar_close(ret);
    return NULL;
}

ssize_t r = xar_read_fd(XAR(ret)->

[...]

if( memcmp(cval, toccksum, tlen) !
    fprintf(stderr, "Checksums do
    free(toccksum);
```

```
        free(cval);
        xar_close(ret);
        return NULL;
    }
```

This first retrieves the attribute checksum attribute from the XML document as a `const char *value`. Then, `strtoull` converts it to an unsigned 64-bit integer and it gets stored in the `offset` variable.

For obtaining the TOC hash for validating the signature, a similar bit of code is used: https://github.com/apple-oss-distributions/xar/blob/f67a3a8c43fdd35021fd3d1562b62d2da32b4f4 L276

```
uint32_t offset = 0;
xar_t x = NULL;
const char  *value;

// xar 1.6 fails this method if an
// within OS X we use this method
// so this method checks and sets

if( !sig )
  return -1;

x = XAR_SIGNATURE(sig)->x;
```

```
/* Get the checksum, to be used fo
   in the future, all checksums sho
if(length) {
  if(0 == xar_prop_get_expect_notn
    *length  = strtoull( value, (c
  }

  if(0 == xar_prop_get_expect_notn
    offset  = strtoull( value, (ch
  }

  if(data) {
    *data = malloc(sizeof(char)*(*

    // This function will either r
    if (_xar_signature_read_from_h
      return -1;
  }
}
```

Note here the tiny but very important difference: while the first comparison was storing the offset in `uint64_t offset` (a 64-bit unsigned integer), here it uses an `uint32_t offset` (a 32-bit unsigned integer). This difference means that **if the offset is outside of the range that can be stored in a 32-bit value, the two checks can use a different heap offset**. For example, if the offset is equal to `0x1 0000`

`0000`, then the integrity hash will be read from `0x1 0000 0000`, while the signature hash will be read from offset `0x0` on the heap.

Thus, it was possible to modify a xar file without invalidating its signature as follows:

> Take a correctly signed xar file and parse the TOC.
> Change the checksum offset value to 4294967296 (and make any other changes you want to the included files, like adding a malicious preinstall script or replacing the installation check script).
> Write the modified TOC back to the file and compute the new TOC hash.
> Add padding until the heap is exactly 4294967296 bytes (4 GiB) in size.[1]
> Place the new TOC hash at heap offset 4294967296, leaving the original TOC hash at heap offset 0.

When this package is verified, the integrity check will use the hash at offset 4294967296, while the signature verification will read it from offset 0. The integrity check will pass, because the new TOC hash is placed there, while the signature will also pass, because the signatures still correspond to the old TOC hash.

Exploitation

This was quite an interesting bug that could be applied in a number of different ways, with different requirements and impact.

Bypassing SIP's filesystem restrictions

When a package is installed that is signed by Apple, installation works a little differently compared to an installation of a package by signed by anyone else. These installations are performed by `system_installd`, instead of `installd`, which has an entitlement granting it access to all files normally protected by SIP:

```
[Key] com.apple.rootless.install.h
[Value]
  [Bool] true
```

This makes sense, as updates from Apple often need to write to protected locations, like replacing components of the OS.

Abusing this vulnerability to modify a package signed by Apple would make it possible to read and write to all those SIP protected files. This could be used to, for example:

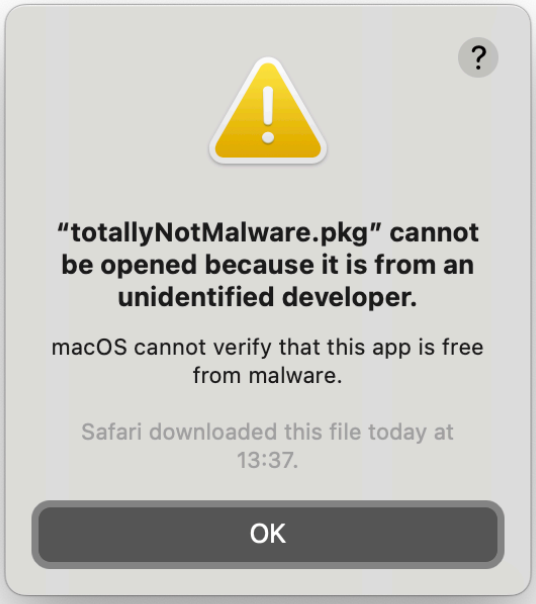
> Grant an application TCC permissions, like access to the webcam, microphone, etc.
> Read data from a data vault, such as the user's Mail and Safari data.
> Load a kernel extension without user approval on Intel macs (although the kernel extension would need to be properly signed).

This could be used to modify a package that a user installs manually, although that requires convincing the user. Another option would be a process that has already obtained root privileges using this to gain access to SIP protected locations, as the root user is allowed to use the `install` command to perform the installation of new packages.

Note that any files on the Signed System Volume (SSV) could not be modified this way, as that disk is mounted read-only.

Bypassing Gatekeeper

After downloading a .pkg file, Gatekeeper will perform a notarization check, similar to that for applications. It takes the hash of the package and submits it to Apple to check that it has been scanned for malware. When a user opens a package that was not notarized, they receive a scary warning, making it quite difficult to trick a user into installing a package containing malware.



The method for querying Apple's server for the notarization status of a package uses the same function to obtain the TOC hash as was used for the signature verification. Therefore, a modified package will still be considered notarized if the original was. This means that if a user downloads such a modified package file, they will not be warned in any way.

Asking users to download a 4 GiB .pkg sounds like a challenge. Even if users don't notice the unusual size, the fact that they need to wait a few minutes for the download to finish could allow them to spot that something is off about the webpage offering the download. Luckily, the padding in the package can be anything, so when using the same byte for all the padding, the resulting file compresses very well. By

placing the package on a compressed disk image, the resulting .dmg file can be only a few hundred kilobytes. Distributing an application in this way is also not unusual for macOS. The increase in size also does not increase the time required for verify the package, as mentioned only the integrity of data on the heap that is actually in use is checked.

Combining this with the previous vulnerability would allow for some very powerful malware: it would be possible to create a manipulated installer package that appears completely legitimate and triggers no warnings when installed. After the user installs it, the malware immediately gains complete access to all SIP-protected data on the system.

Elevating privileges

We did not find a way to abuse this vulnerability for privilege escalation on an out-of-the-box installation of macOS. However, when combined with certain third-party software, we did find a method.

Some applications try to make sure that their application can update itself automatically, even if the current user is not an admin user. Normally, non-admin users are not allowed to make changes in `/Applications`, so they can not update any existing applications. If the admin never logs in, this could mean that

users run known vulnerable software indefinitely.

To solve that, some applications include a privileged helper tool to perform the upgrade. This is a tool that runs as root and has the single purpose of installing updates for the existing application. Often, the application itself handles the checking for updates and downloading a new update file, the tool only performs the actual installation.

To make this secure, there are two important checks:

> A request to install an update must originate from the associated application.
> The update file must be authentic (and not a downgrade).

The format of the update file varies between the applications that implement this, but using .pkg files is common. If this method is used, then it may be possible to swap out an update package with a modified version. For example, by using a race condition to change the package in between the download by the application and the actual installation by the privileged helper tool. This means that the package would be installed automatically, allowing privilege escalation to root.

In fact, this vulnerability was originally

discovered when investigating the privileged helper tool used by Zoom. In the DEF CON 30 talk "You're Muted Rooted" by Patrick Wardle he described a method for bypassing the signature verification performed by Zoom. This was addressed by switching to the libxar functions for verifying a package signature.

Non-impact

During our research to investigate the full impact of this vulnerability, we also attempted to modify macOS system updates. These also use .pkg files and verify the TOC hash, however, they compare it to the computed TOC hash. Therefore, replacing a system update with a malicious file is not possible.

This issue also does not affect iOS, as xar files are not used there anywhere as far as we could tell. While signed xar files have been used for Safari extensions in the past, they now use app extensions, so we could also not identify any impact there.

Demo

The following video demonstrates the use of this vulnerability to bypass Gatekeeper and SIP. As can be seen, it creates a new file in `/private/var/db/SystemPolicyConfiguration/`, a directory normally protected by SIP.

(Note that the installer states that the installation has failed, but the exploit already ran using a pre-install script. This is only the case for the demo and could be avoided for a real attack.)

The fix

This was fixed by Apple with a 2 character fix: changing `uint32_t` to `uint64_t` in macOS 13.1.

What is interesting about this vulnerability is that there was a similar issue in 2010: CVE-2010-0055. In that version, one of the checks assumed that the TOC hash offset was always 0 and the other used the value read from the TOC. Vulnerabilities that are variants of fixed issues and regressions that re-introduce a vulnerability are sadly common, but to see a vulnerability similar to a 12 year old vulnerability is still surprising. Especially considering that a small change to this library could prevent all similar vulnerabilities that

lead to the same result.

A comment in the code snippet above notes the following:

> *Store our toc hash upon archive open, so callers can determine if it has changed or been tampered with after archive open*

Using this stored value instead of reading it from the file again would have made this vulnerability, and any similar variants, impossible to exploit as the value would not be read from the heap twice.

> If the original package is already more than 4 GiB in size, then there are a number of options. For example, padding the file to 8, 12, 16, etc. GiB instead. Or it would be possible to move files on the heap around to make the offset 4294967296 available. ↩

BACK