SensePost | Outlook Forms and Shells

Archived: 2026-04-05 17:07:58 UTC

Using MS Exchange and Outlook to get a foothold in an organisation, or to maintain persistence, has been a go to attack method for RedTeams lately. This attack has typically relied on using Outlook Rules to trigger the shell execution. Although Ruler makes accomplishing this really easy, it has, up until now, required a WebDAV server to host our shell/application. In most cases this is not an issue, but once in a while you run into a restrictive network that does not allow the initial WebDAV connection to be established. In such instances, the attack sadly fails. Another downside to Outlook rules is that we are limited to only providing an application path, and no command-line arguments, meaning none of our fancy Powershell one-liners can be used.

Lastly, Microsoft has released a patch for Outlook 2016 that disables both our *Run Application* and *Run Script* rules by default.

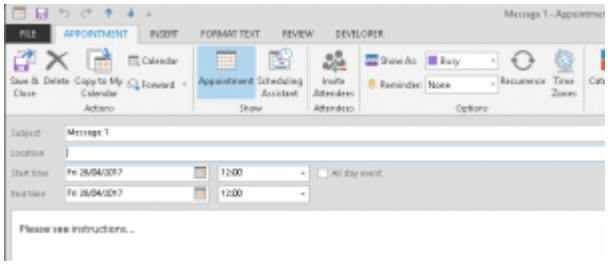
It was time to find a new attack avenue.

Attack Surface

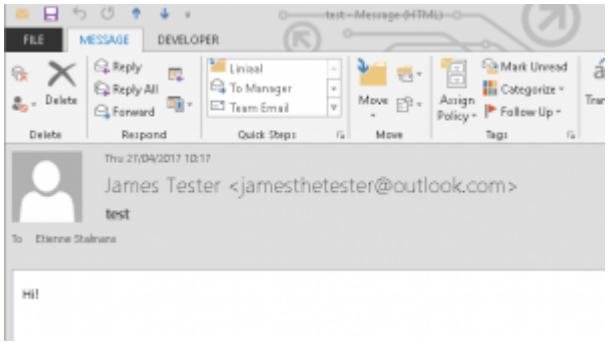
I set out to try and find another way to get a shell through Outlook, in the case of us having valid credentials. The first interesting angle would be to use the VBA Macro engine built into Outlook. Unfortunately, this is a no go for a few reasons. Firstly, VBA Macros are not synchronised between Outlook instances, unlike rules. Secondly, as mentioned above, *Run Script* rules are not going to be available going forward. And lastly, more and more organisations are (finally) moving towards a “block all macros” policy.

Fortunately for us, Outlook has a massive attack surface and provides several other interesting automation features. One of these is *Outlook Forms*. Forms provide a user/organisation with email customisation options on how it is presented or composed. This includes features such as autocompleting the *bcc* field or inserting template text.

All Outlook message objects are actually forms in their own right. The only difference between an *Appointment Request* message and a normal *Message*, is the form used to display these in the Outlook UI.



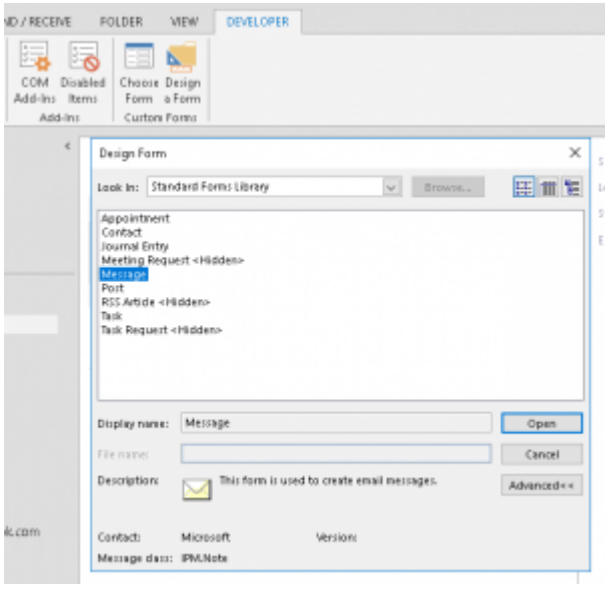
Appointment Message, using IPM.Appointment



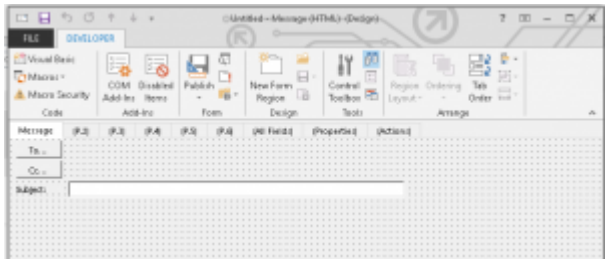
Message form, your standard IPM.Note

Now, this is interesting: you can change the way a message appears or what fields are available to a user when composing a new message. More information about forms can be viewed directly from the source, Microsoft: https://msdn.microsoft.com/en-us/library/office/ff868929.aspx

What if you want to create your own form? It is pretty simple, you need to enable the *Developer* tab in the ribbon, and select *Design a Form*. This opens a form designer where you can move and edit various form fields.

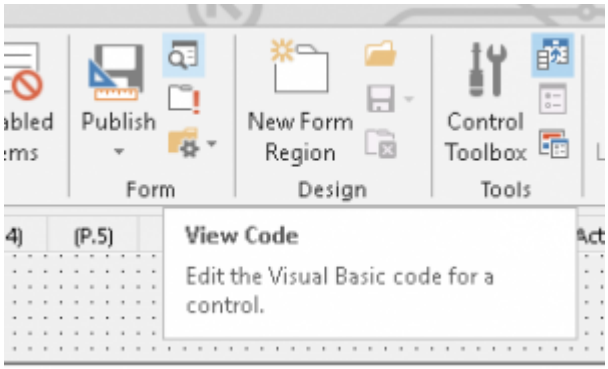


Selecting a form to design. This can be based off of existing forms.



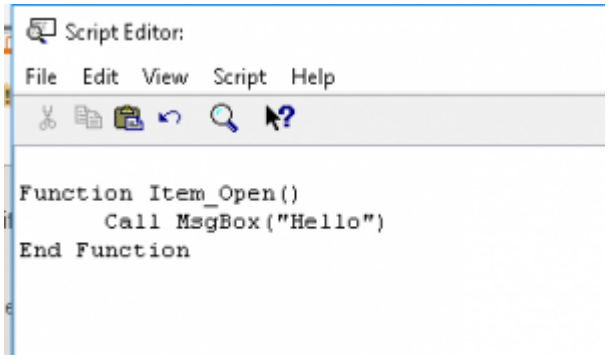
The form designer.

While looking at this designer, an interesting little area caught my attention. If you hover over the icon with a form and magnifying-glass you will get a pop-up message *View Code – Edit the Visual Basic code for a control.* You may now figure where this is going…



The option to edit VBScript

Once you open up the code viewer, you will see a very basic text editor with two options under the *script* menu, namely *Event Handler* and *Object Browser*. That is it, you need to figure out the rest by yourself (unlike the nice VBA editor that ships with Office). Selecting the *Event Handler* pick the *open* option and VBScript will be inserted to handle the *on_open* event for this form.



Creating some script

If you close the script editor, you can now run the form. This is done by using the *Run this form* button, found right below the *View Code* button. Immediately a MsgBox will pop up, along with the new form!

I had disabled Macros in Outlook, so how could this code be running?!

It turns out that this script engine is separate from the VBA Macro script engine, as explained here: http://drglennn.blogspot.co.uk/2008/07/vbscript-and-custom-forms-in-outlook.html.

This means, we have a full VBScript engine available to us, and can now start trying to insert a more juicy payload.

A test payload of:

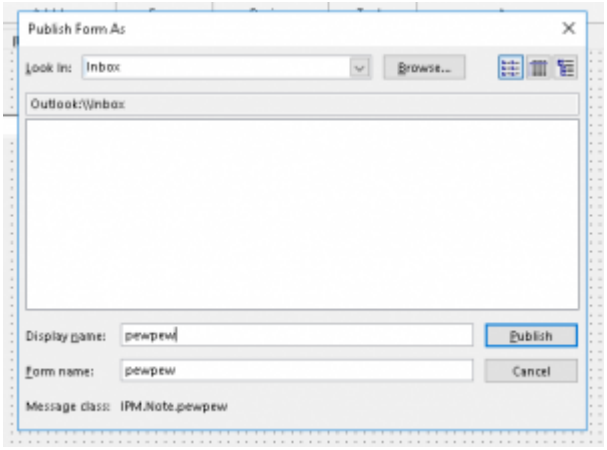
```
Function Item_Open()
     CreateObject("Wscript.Shell").Run "calc.exe", 0, False
End Function
```

Yields a nice calculator as soon as the form is opened. This can be extended to perform different actions when a reply is created, a message is forwarded, etc. Winning, so far. Now the big test, does it persist and does it

synchronise?

Save a form

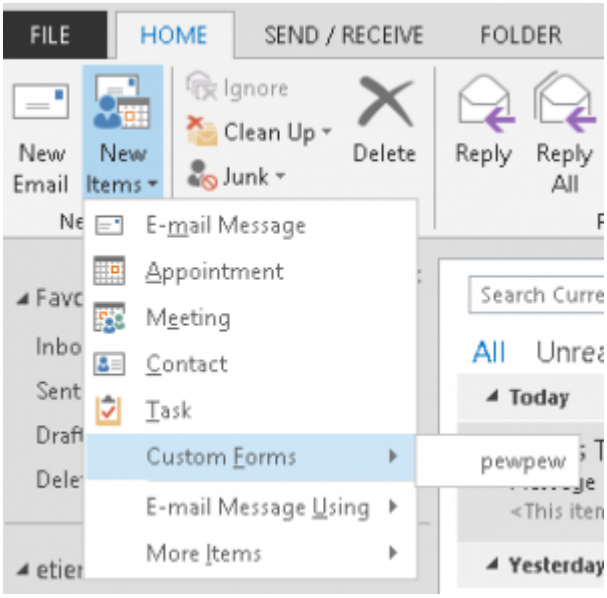
Forms can either be saved to a local file (Save-As) which then makes it available to the local install of Outlook, not very useful for us, but you can also *Publish* a form. When publishing, the form can be stored in the *Personal Forms Library* or in an Outlook folder, such as the Inbox.



The form can be published to an Exchange folder.

Once published, the form gets a new form name and its own message class. The message class is important, as Outlook will use this to determine which form to use when processing a message and displaying it. **If the form gets published into a folder such as the Inbox, it automatically gets synchronised with the Exchange server, and all instances of Outlook associated to that account!**

After the form is saved, you need to be able to activate/trigger it somehow. This can be done directly through the Outlook interface, but this is not very useful, as you will be shelling yourself. If you want to test if a form works as you would like it to, select the inbox and then in the ribbon select *New Items* -> *Custom Forms* -> Form_name.



The new form can be accessed through the menu.

How about triggering this remotely? For this you need to send an email of the correct message class. To do this through Outlook, you will need to create a form with the same message class as the one you have created on your target's mailbox, and then send an email using that form. Convoluted, I know.

Automating – Ruler

As demonstrated by the last example of how to trigger the email, it is rather complicated trying to do this through Outlook. Also creating the malicious form in Outlook, would require you to synchronise the mailbox to your own host. This seemed an ideal feature to extend the functionality in Ruler and create these forms automatically.

No Documentation

The first issue in doing this was figuring out where or how these forms are actually stored. When developing Ruler I was fortunate enough to have specific MAPI Remote Operations (ROPS) available to interact with the Rules Organiser in Exchange. Sadly, with forms I was not so lucky. Turns out forms do not have their own ROPS, and documentation of how these are created/stored/accessed through MAPI is near non-existent, as far as I know.

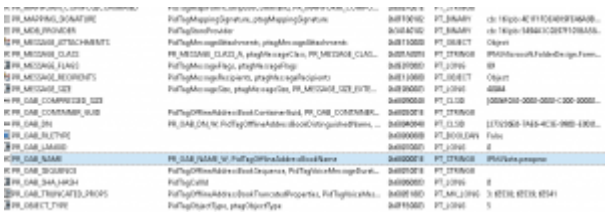
Fortunately, there is a brilliant application we can use to view a Mailbox and all MAPI objects associated with that mailbox. **MFCMAPI**, tool that provides access to MAPI stores to facilitate investigation of Exchange and Outlook issues – https://github.com/stephenegriffin/mfcmapi – if you are serious about looking into MAPI development, this is the tool to turn to.

Discovery

Using MFCMAPI I was able to determine that forms get stored in the *associated* table for a folder. The associated table is described as *"A collection of Message objects that are stored in a Folder object and are typically hidden from view by email applications. A FAI Message object is used to store a variety of settings and auxiliary data, including forms, views, calendar options, favourites, and category lists."*

Looking at the associated folder using MFCMAPI we can find the form message. This is saved with a message class of **IPM.Microsoft.FolderDesign.FormsDescription**. Each form is then described in the attachments and P*roperyTags* stored with these messages. There are a bunch of options that need to be set for each form, and these are controlled through the *PropertyTags* of the message.

Weirdly enough, *PropertyTags* such as **PidTagOfflineAddressBookName** are used to describe the form, there aren't any **PidTag[form]** *PropertyTags*. This particular tag controls the message class of the form – in this case **IPM.Note.pewpew**. The form document, along with the VBScript to execute is saved in an unique attachment, the data of this form is stored in the Outlook message format, basically the same format used when you save a message to disk.



Inspecting the associate table with MFCMAPI

Now that I knew where the forms where stored, I had to figure how to create them. The MAPI library I created for Ruler had all the functions needed to create messages, attachments and custom *PropertyTags*. The complicated part was deciphering exactly which tags are required and what the different values mean (we want dynamic forms after all). This took a few painful hours of creating messages and comparing the Ruler created message with that generated in Outlook. Eventually I stumbled on all the required values.

It also allowed for the discovery of some nice "hidden" features. For example, setting the **PidTagSendOutlookRecallReport** to **true** would hide the form from the user interface. This means the new form won't show up under *custom forms* in the *new item* menu. To discover the new form, a user would need to go into the advanced options tab in Outlook, navigate to forms, select the inbox and view the list of forms (unlikely).

The inbox is also not the default forms library location, thus users who do modify forms, tend to only see those stored in the Personal Forms Library, which is the default view and storage space.

Custom Form

Being able to create a form through Ruler was great, but how about specifying a custom payload, custom VBScript? This got a little complicated.

Option one was to sift through the MS-OXMSG.pdf and create a custom parser/generator for the MSG format; option two was to hexedit an existing form. I went for the latter.

The current form script template in use:

```
Function P()
>>MAGIC<<
End Function

Function Item_Open()
Call P()
End Function



Function Item_Reply(ByVal Response)
Call P()
End Function



Function Item_Forward(ByVal ForwardItem)
Call P()
End Function
```

```
Function Item_Read(ByVal Response)
Call P()
End Function
```

When Ruler generates the new form, it searches through form template for our “MAGIC” token and replaces this with the supplied payload. For now, this means that the payload size is limited to 4096 bytes, which should be more than enough for creating a useful payload. This means the standard base64 encoded Empire launcher should fit. But we are hackers, with 4096 bytes you should be able to do A LOT of interesting things.

You should also notice that there are multiple triggers for this form. The payload will be called if the message is read (previewer), opened (not previewed) or if the user attempts to reply or forward the message. This means that the user needs to at least **preview** the message. Alternatively you need a slight amount of social engineering, where the attacker needs to either get the user to open the message or to reply to it. A nice side affect is that the user will inadvertently trigger the payload if they try “forward” it to the incident response team.

Attack Attack

A few changes to Ruler and the new attack is baked in. The new functionality can be accessed through the **form** command.

```
./ruler --email john@msf.com form help

USAGE:
ruler form [global options] command [command options] [arguments…]


VERSION:
2.0.17


COMMANDS:
add creates a new form.
send send an email to an existing form and trigger it
delete delete an existing form
display display all existing forms
```

Under the **form** command, there are a number of sub commands. Just as you have with standard Ruler. To add a new form:

```
./ruler --email john@msf.com form add --suffix pewpew --command "MsgBox(\"hello\")" --send
```

In this example, this will create a new form, with the message class **IPM.Note.pewpew** and the VBScript to show the MsgBox. The command can also be supplied from a file, using the *–input* argument.

Leaving out the *–command* or *–input* arguments will provide with a sample VBScript to use.

The *–send* specifies that we want a trigger mail to be sent. The default email has a subject of "Invoice [Confidential]" and the body "This message cannot be displayed in the previewer". If you wish to customise this email, simply use –subject "new subject" and –body "new body".

To trigger an existing form, you can use the **send** command.

```
./ruler --email john@msf.com form send --suffix pewpew
```

And same story with the subject and body. Forms can also be retrieved and deleted. If you wish to view a list of current forms, use the **display** command and to delete a form, **delete –suffix pewpew**.

The end result of this can be seen here: https://youtu.be/XfMpJTnmoTk

Code

The new version of Ruler is available on Github: https://github.com/sensepost/ruler

NOTE: If you are going to use the pre-built binaries, there is an extra setup step!

In your current working directory, you need a folder named "templates". In this folder you will require the files:

- img0.bin
- img1.bin
- formtemplate.bin

These can be retrieved from the github folder: https://github.com/sensepost/ruler/tree/master/templates

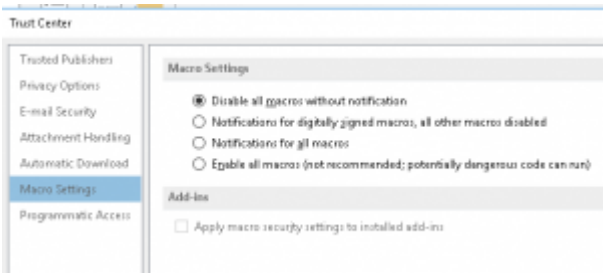
Persistence

What I really like about this method is the fact that it is ideal for persistence. If you compromise an account, install a custom form and that is it. If you lose access to that account, all you need to do to trigger the shell again is to send an email with the correct message class. Unlike rules, there is not an easy way in the UI for a user to check their forms. And as far as I can see, forms can not be seen in OWA, unlike Rules. Because the message is saved in the associated table, the standard query tools for Exchange do not allow you to retrieve this information.

Defence

Apart from MFA/2FA, defending against this is going to be interesting, just like Rules, I cannot give a firm "this will block it" solution.

Good monitoring and logging should be able to pickup the VBScript execution. Unfortunately, based on my testing, disabling macros does not protect against this. I have not tried with macros disabled via GPO, but with *Disable all macros without notification* set in Outlook, the VBScript will still execute.



This setting doesn't seem to help.

If there are other ways of blocking VBScript in Outlook, please share and I will update this post.

Future Work

Depending on time and the popularity of this specific method of getting shell, I will look into better customisation options for the supplied script. This is going to take some more work, but the idea is to implement some logic that allows the attacker to choose when the script should trigger, either on open, reply, forward, etc, or all of them.
