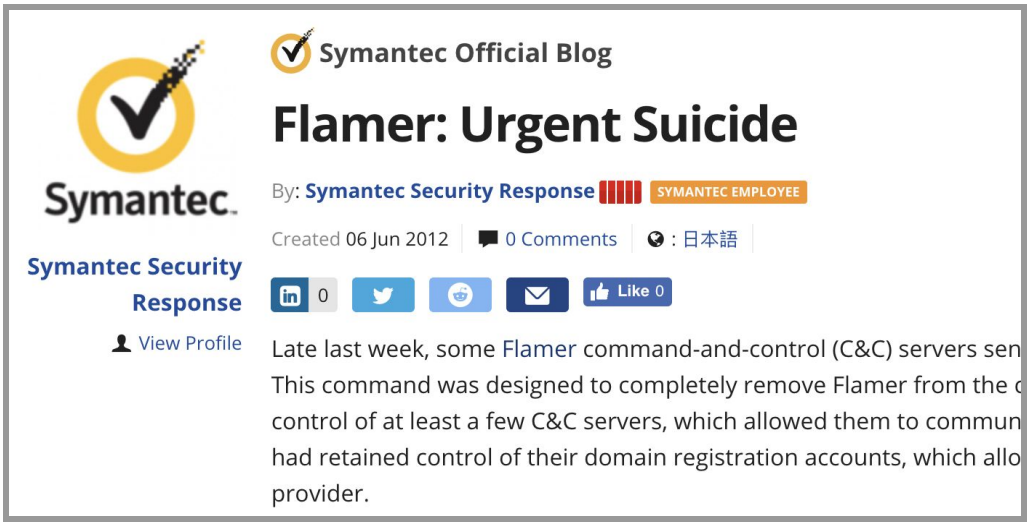
Flame 2.0: Risen from the Ashes

9 April 2019 – J. A. Guerrero-Saade (turla@Chronicle.Security)
Silas Cutler (havex@Chronicle.Security)

Our investigation into the GOSSIPGIRL Supra Threat Actor (STA) started with a REPLICANT FARM[1] signature name that tentatively links the cryptonym GOSSIPGIRL to Flame. From there, we investigated MiniFlame and Gauss –two families related to the Flame platform– without finding any indication of succession to Flame's operations. Our investigation continued onto Stuxnet and Duqu but the altogether disappearance of Flame never sat right with us.



Flamer: Urgent Suicide (Symantec)[2]

Sometime in late May 2012, researchers witnessed an attempted cleanup of the remaining Flame infections. Whatever command-and-control servers the attackers still had control of (as many had been sinkholed by researchers) began to distribute a SUICIDE module and were then themselves scrubbed. The SUICIDE module was a particularly clunky way of burning down operations as it gave researchers a full list of the components and directories that the operators sought to delete - possibly something they rushed to develop. By all accounts, this was considered the death of Flame.

However, during our investigation into the GOSSIPGIRL Supra Threat Actor (STA) cluster, we found ourselves wrestling with Flame once again, in a way we hadn't expected.

---

[1] https://edwardsnowden.com/docs/doc/media-35688.pdf
[2] https://www.symantec.com/connect/blogs/flamer-urgent-suicide

Flame in Context

In May 2012, the Iranian CERT MAHER, Kaspersky Lab, and Crysis Lab researchers announced the discovery of an advanced malware platform targeting Iran. Flame (a.k.a. Flamer or sKyWIper) surprised researchers with its sheer size and the breadth of its capabilities, as an all-in-one cyberespionage toolkit. While on it's own Flame was groundbreaking, later research identified links that showed Flame and Stuxnet were directly connected. If modularity and extensibility have become an expected architectural feature in modern APT toolkits, Flame was one of the seminal modular platforms.

The malware packs a Lua virtual machine, allowing it to execute a series of scripts to implement complex capability with relative ease. These modules enabled the operators to gather system information, beacon to nearby bluetooth devices, implement network replication, propagate to other machines or removable media, create backdoor accounts, and much more. The malware gained true recognition with the discovery that one of its submodules (named 'GADGET') used valid Microsoft digital certificates generated by abusing an MD5 collision attack in order to spread across an enterprise via the Windows Update mechanism.

An Enduring Mistake

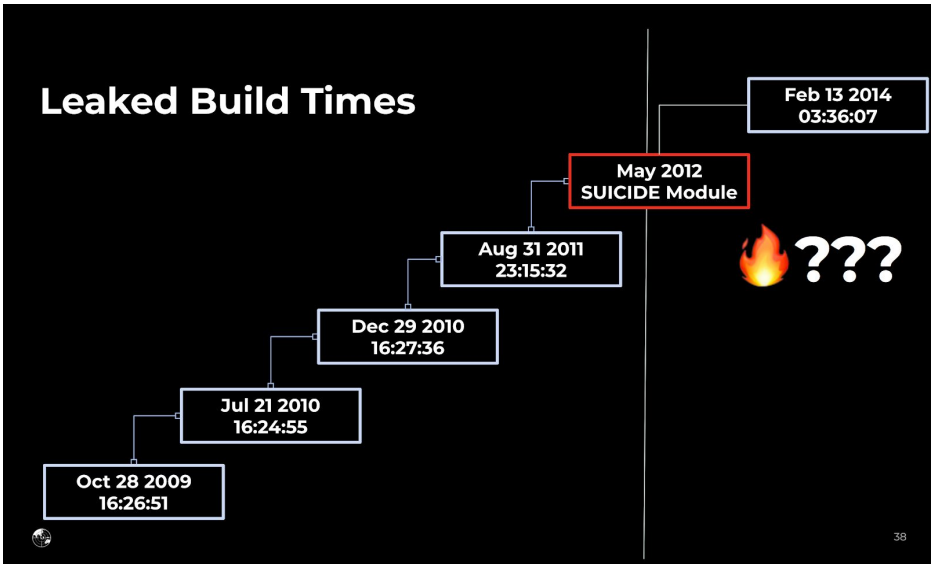
Most Flame samples are obviously timestomped to hide their real compilation time. However, as Crysys Lab researchers noted, when debug symbols are left in the samples, they leak the underlying compilation timestamp of a statically-linked library. The Crysys researchers thought this was SQlite but in our assessment, it appears to be PuTTY[3]. When a version isn't clearly identified, the following string is printed:

```
32
33  char ver[] = "Unidentified build, " __DATE__ " " __TIME__;
34  char sshver[] = "PuTTY-Local: " __DATE__ " " __TIME__;
35
```

PuTTY Source Code, version.c file.

This allowed us to track an earliest possible date of compilation for different Flame components. Most of the results were as expected, in the October 2009 to August 2011 range.

[3] https://github.com/sztupy/adbputty/blob/master/VERSION.C



Tracking leaked build times in reference to the deployment of the FLAME suicide module

However, a subset of samples surprised us. Their leaked build times pointed to a range of February-March 2014, nearly two years after Flame operations were burned down and the platform was considered abandoned.

Looking at these samples lead us to the discovery of a new iteration of the Flame platform, likely used in the 2014-2016 timeframe. While the malware is clearly built on the Flame source code, it includes new counter-measures against researcher meddling. We hope that announcing these findings at an early stage will encourage a collaborative environment in the threat intelligence space reminiscent of the early days of discovery that brought about Stuxnet, Duqu, Flame, and Gauss.

Technical Analysis

Much like the original Flame, Flame 2.0 continues to comprised of multiple submodules directed by a main orchestrator reliant on an embedded Lua VM. Our initial hunting revealed two new sets of samples that we consider orchestrators orchestrators (sensrsvcs / sensrsvr) and likely submodules (wmisvcs / wmihost).

Flame 1.0 modules were stored in embedded resources decoded via an XOR-based cipher and then ZLIB decompressed. In the newer iteration, the authors adopted AES-256[4] to encrypt the embedded resources. Unlike other developers who embed their decryption keys alongside the data, the operators are expected to pass the necessary decryption key to the orchestrator upon execution in the form of an argument to the various DLL exports. As a result, the contents of

[4] This version also supports AES-128 and AES-192

these resources remain unknown. We hope that additional samples, heuristic execution logs, and overall greater visibility into the campaign will eventually enable us to analyze the contents.

Flame 2.0 Orchestrator Samples

SHA256	File Name	Description	Detections[5]
15a9b1d233c02d1fdf80071797ff9077f6ac374958f7d0f2b6e84b8d487c9cd1	sensrsvcs.dll	- x64 - Contains Lua 5.1 controller - Resources 101, 102, 103, 104, 105	1/57
426aa55d2afb9eb08b601d373671594f39a1d9d9a73639c4a64f17d674ca9a82	sensrsvcs.dll	- Contains Lua 5.1 controller - Resources 101, 102, 103, 104, 105	5/57
af8ccd0294530c659580f522fcc8492d92c2296dc068f9a42474d52b2b2f16e4	sensrsvr.dll	- Contains Lua 5.1 controller - Resources 101, 102, 103, 104, 105	5/57
69227d046ad108e5729e6bfaecc4e05a0da30d8e7e87769d9d3bbf17b4366e64	sensrsvr.dll	- x64 - Contains Lua 5.1 controller - Resources 101, 102, 103, 104, 105	1/57

Additionally, we've identified the following related samples. We suspect that these are submodules possibly dropped by the orchestrators from the encrypted resources.

Flame 2.0 Suspected Submodules

0039eb194f00b975145a35ede6b48d9c1ea87a6b2e61ac015b3d38e7e46aecbb	wmisvcs64.dll	- x64 - Embedded Plink - Build after 13 Feb 2014 03:40:40	0/57
8cb78327bd69fda61afac9393187ad5533a63d43ebf74c0f9800bedb814b20ad	wmisvcs64.dll	- x64 - Embedded Plink	0/57

[5] Detections were noted at time of writing and are subject to improvement. These numbers represent static detections and may not accurately represent heuristic detections at sample execution.

		- Build after 13 Feb 2014 03:40:40	
b61c62724421d38a13c58877f31298bd663c1c8f8c3fe7d108eb9c8fe5ad0362	wmihost64.dll	- x64 - Build after 30 Mar 2014 18:15:53	0/57
134849f697ab5f31ffb043b06e9ca1c9b98ffebba8af8ccdedd036a6263bf3a4	wmihost.dll	- Build after 30 Mar 2014 18:15:53	2/57

Discernible Functionality

While being unable to decode the embedded modules kept us from determining large parts of the functionality, some of Flame 2.0's capabilities are still discernible. Deobfuscated strings contain hints and recognizable references. Python decryptors are included in the Appendix.

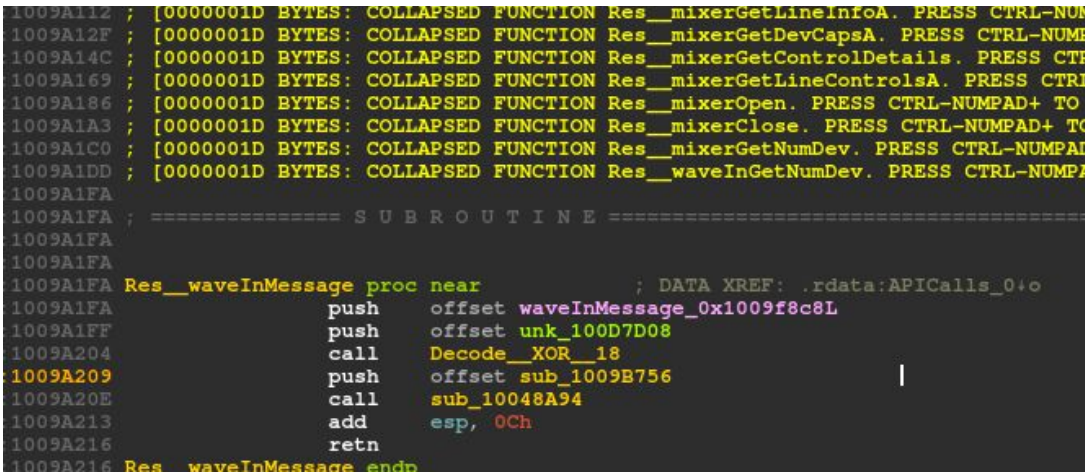
Flame 2.0 resolves the requisite Windows API calls dynamically during execution. The names of these functions are decoded, resolved using LoadLibrary() and GetProcAddress(), and loaded into a struct. An example of one of the core structures used during the initialization of a submodule[6] is shown below:

```
struct CallStruct_0x1002911F
{
  int UNK;
  int GetProcAddress;
  int *GetModuleHandleA;
  int *GetLastError;
  int *memcpy;
  int *OpenFileMappingW;
  int *CreateFileMappingW;
  int *UnmapViewOfFile;
  int *MapViewOfFile;
  int *LoadLibraryW;
  int *LoadLibrary;
  int *FreeLibrary;
  int *NtQueryInformationProcess;
  int *CloseHandle;
  int *CreateMutexW;
  int *OpenMutexW;
  int *ReleaseMutex;
  int *VirtualProtect;
  int *VirtualFree;
  int *VirtualAlloc;
```

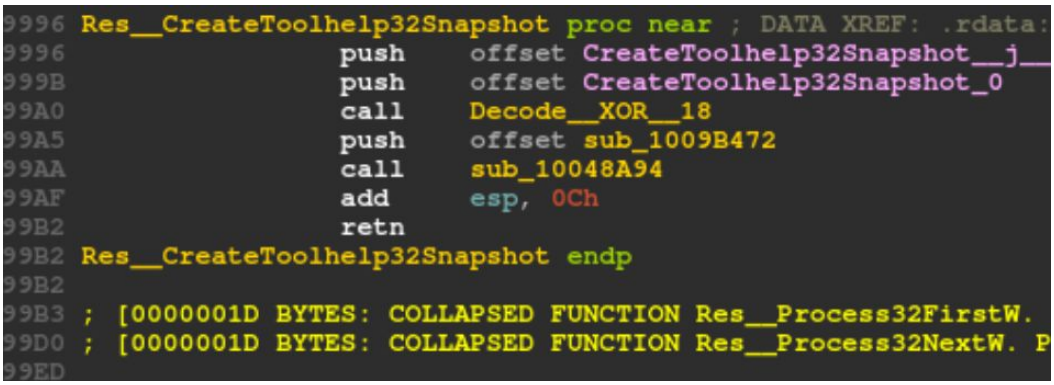
[6] SHA256: 0039eb194f00b975145a35ede6b48d9c1ea87a6b2e61ac015b3d38e7e46aecbb

```
  int *WaitForSingleObject;
  int *CreateFileW;
  int *memset;
  int *Sleep;
  int *LocalFree;
};
```

The resulting API calls may point in the direction of Flame 2.0's capabilities, just as they may simply support underlying execution requirements. For example, API calls loaded in order to enumerate processes are used by Flame2.0 to check for the existence of certain antivirus products. However, future analysis will likely find process enumeration also used for generating a system profile. Some of the more illustrative uses of Windows API calls are shown below alongside their more overt functionality:



Interacting with infected host's audio input



Process enumeration

Additionally, based on Putty[7]-related debugging strings and embedded artifacts, it's likely that Flame 2.0 is designed to support lateral movement. The following decoded strings match the syntax used by PLINK for remote port forwarding:

```
-pw %s -R %d:127.0.0.1:%d -N %s@%s -P %d

-pw %s                  - Password
-R %d:127.0.0.1:%d      - <ListenPort>:<LocalHost>:<LocalPort>
-N                      - Suppress starting a shell
%s@%s                   - User@Host
-P %d                   - Port
```

A Call For Collaboration

We hope that releasing these indicators at an early stage in our research process will encourage collaboration from the Threat Intelligence community. At this time, Flame 2.0 remains largely a mystery but one that's likely to reward collective research.

[7] https://www.putty.org/

Appendix

String Decryption

```
def DecodeMethod1(indata, r_start, r_length):
	dec_data = ""
	enc_data = indata[r_start:]
	dec_len = ord(indata[r_length])

	for index, byte in enumerate(enc_data[:dec_len]):
		eax = (
			(((dec_len - index) -1) ^ 0x1D) * ((dec_len - index) + 0x10)
			) & 0xFFFFFFFF
		eax += 0x1000193
		cl = ( ((eax >> 0x18) & 0xFF) ^ ((eax >> 0x10) & 0xFF) )
		cl = ( cl ^ ((eax >> 0x8) & 0xFF) )
		cl = ( cl ^ ord(byte)) & 0xFF
		cl = ( cl ^ (eax & 0xFF) )
		dec_data += chr(cl)

	return dec_data
```

```
def DecodeMethod2(indata, key, r_start, r_length):
	enc_data = indata[r_start:]
	dec_length = ord(indata[r_length])
	dec_data = ""
	for index, byte in enumerate(enc_data[:dec_length]):
		if ord(enc_data[index]) == 0 and ord(enc_data[index+1]) == 0: break
		dec_data += chr( ord(byte) ^ ord(key[index % len(key)]) )

	return dec_data
```

System Artifacts

Path	Suspected Use
%TEMP%\tmpinstall.tmp	Log FIle
Global\ComConnectEvent	Global Event Name
\\.\pipe\ComConnect	Named Pipe
Global\WMI_CONNECTION_RECV	Global Event Name
HKLM\Software\Microsoft\DirectXHelp\	Unknown
HKLM\Software\Microsoft\HelpDirectXRep\	Unknown

Yara Rules

```
import "pe"
import "hash"
rule FLAME2_Orchestrator
{
	meta:
		desc = "Encrypted resources in Flame2.0 Orchestrators"
		author = "turla @ Uppercase"
		hash1 =
		"15a9b1d233c02d1fdf80071797ff9077f6ac374958f7d0f2b6e84b8d487c9cd1"
		hash2 =
		"426aa55d2afb9eb08b601d373671594f39a1d9d9a73639c4a64f17d674ca9a82"
		hash3 =
		"af8ccd0294530c659580f522fcc8492d92c2296dc068f9a42474d52b2b2f16e4"
	condition:
		for any i in (0..pe.number_of_resources - 1):
			((hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"53b19d9863d8ff8cde8e4358d1b57c04") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"4849cc439e524ef6a9964a3666dddb13") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"62bfe21a8eb76fd07e22326c0073fef5") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"dfed2c71749b04dad46d0ce52834492c") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"9119aa701b39242a98be118d9c237ecc") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"b69d168e29fba6c88ad4e670949815aa") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"4849cc439e524ef6a9964a3666dddb13") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"1933a1e254b1657a6a2eb8ad1fbe6fa3") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"dfed2c71749b04dad46d0ce52834492c") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"9119aa701b39242a98be118d9c237ecc") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"b69d168e29fba6c88ad4e670949815aa") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"17c794f7056349cb82889b5e5b030d15") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"e15187f79b6916cb6763d29d215623c1") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"923963bb24f2e2ceac9f9759071dba88") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"9a2766aba7f2a56ef1ab24cf171ee0ed") or
			(hash.md5(pe.resources[i].offset, pe.resources[i].length) ==
			"ebe15bfb5a3944ea4952ddf0f73aa6e8"))
}
```