CARBANAK Week Part One: A Rare Occurrence

fireeye.com/blog/threat-research/2019/04/carbanak-week-part-one-a-rare-occurrence.html



It is very unusual for FLARE to analyze a prolifically-used, privately-developed backdoor only to later have the source code and operator tools fall into our laps. Yet this is the extraordinary circumstance that sets the stage for CARBANAK Week, a four-part blog series that commences with this post.

CARBANAK is one of the most full-featured backdoors around. It was used to perpetrate millions of dollars in financial crimes, largely by the group we track as <u>FIN7</u>. In 2017, Tom Bennett and Barry Vengerik published <u>Behind the CARBANAK Backdoor</u>, which was the product of a deep and broad analysis of CARBANAK samples and FIN7 activity across several years. On the heels of that publication, our colleague Nick Carr uncovered a pair of RAR archives containing CARBANAK source code, builders, and other tools (both available in VirusTotal: <u>kb3r1p</u> and <u>apwmie</u>).

FLARE malware analysis requests are typically limited to a few dozen files at most. But the CARBANAK source code was 20MB comprising 755 files, with 39 binaries and 100,000 lines of code. Our goal was to find threat intelligence we missed in our previous analyses. How does an analyst respond to a request with such breadth and open-ended scope? And what did we find?

My friend Tom Bennett and I spoke about this briefly in our 2018 FireEye Cyber Defense Summit talk, <u>Hello, Carbanak!</u> In this blog series, we will expound at length and share a written retrospective on the inferences drawn in our previous public analysis based on binary code reverse engineering. In this first part, I'll discuss Russian language concerns, translated graphical user interfaces of CARBANAK tools, and anti-analysis tactics as seen from a source code perspective. We will also explain an interesting twist where analyzing the source code surprisingly proved to be just as difficult as analyzing the binary, if not more. There's a lot here; buckle up!

File Encoding and Language Considerations

The objective of this analysis was to discover threat intelligence gaps and better protect our customers. To begin, I wanted to assemble a cross-reference of source code files and concepts of specific interest.

Reading the source code entailed two steps: displaying the files in the correct encoding, and learning enough Russian to be dangerous. Figure 1 shows CARBANAK source code in a text editor that is unaware of the correct encoding.



Figure 1: File without proper decoding

Two good file encoding guesses are UTF-8 and code page 1251 (Cyrillic). The files were mostly code page 1251 as shown in Figure 2.

🕼 errors.h (~\Desktop\samples\car\unpack\botep\bot\include) - GVIM6						
<u>File Edit T</u> ools <u>Syntax Buffers Window H</u> elp						
⊖ ⊟ ◘ ≜ 9 @ X @ @ & & & & ≛ ≛ & î 4 ⊶ ? ?						
1 //ошибки которые возвращает бот админке при выполнении команд 2 3 namespace TaskErr	-					
4 { 5 $6 \text{ const int Succesfully} = 0: //Het ошибки$	=					
7 const int Wrong = 1; //команда не выполнилась 8 const int Param = 2; //неверные параметры						
9 const int PluginNotLoad = 3; //не загрузился плагин 10 const int NotCreateDir = 4; //не удалось создать папку						
11 const int Runned = 5; //yже запущено 12 const int OutOfMemory = 6; //нехватка памяти 13 const int NotRunInMem = 7: //не удалось стартануть в памяти						
14 const int NotExportFunc = 8; //не удалось найти экспортируемые функции в длл 15 const int BadCab = 9; //не удалось распаковать сав архив :set enc=cp1251	Ŧ					

Figure 2: Code Page 1251 (Cyrillic) source code

Figure 2 is a C++ header file defining error values involved in backdoor command execution. Most identifiers were in English, but some were not particularly descriptive. Ergo, the second and more difficult step was learning some Russian to benefit from the context offered by the source code comments.

FLARE has fluent Russian speakers, but I took it upon myself to minimize my use of other analysts' time. To this end, I wrote a script to tear through files and create a prioritized vocabulary list. The script, which is available in the <u>FireEye vocab_scraper GitHub repository</u>, walks source directories finding all character sequences outside the printable lower ASCII range: decimal values 32 (the space character) through 126 (the tilde character "~") inclusive. The script adds each word to a Python defaultdict_ and increments its count. Finally, the script orders this dictionary by frequency of occurrence and dumps it to a file.

The result was a 3,400+ word vocabulary list, partially shown in Figure 3.

Figure 3: Top 19 Cyrillic character sequences from the CARBANAK source code

I spent several hours on Russian language learning websites to study the pronunciation of Cyrillic characters and Russian words. Then, I looked up the top 600+ words and created a small dictionary. I added Russian language input to an analysis VM and used Microsoft's on-screen keyboard (osk.exe) to navigate the Cyrillic keyboard layout and look up definitions.

One helpful effect of learning to pronounce Cyrillic characters was my newfound recognition of English loan words (words that are borrowed from English and transliterated to Cyrillic). My small vocabulary allowed me to read many comments without looking anything up. Table 1 shows a short sampling of some of the English loan words I encountered.

692: в
301: для
265: не
269: если
224: файл(а)
20: файле
20: файлы
18: файлов
223: и
203: на
189: c
173: то
145: сервер(а)
22: сервером
10: серверов
14: серверу
140: возвращает
2: вернули
134: имя

Cyrillic	English Phonetic	English	Occurrences	Rank
Файл	f ah y L	file	224	5
сервер	server	server	145	13
адрес	adres	address	52	134
команд	k o m a n d	command	110+	27
бота	bota	bot	130	32

плагин	p l ah g ee n	plugin	116	39
сервис	s e r v ee s	service	70	46
процесс	p r o ts e s s	process	130ish	63

Table 1: Sampling of English loan words in the CARBANAK source code

Aside from source code comments, understanding how to read and type in Cyrillic came in handy for translating the CARBANAK graphical user interfaces I found in the source code dump. Figure 4 shows a Command and Control (C2) user interface for CARBANAK that I translated.

-	bokkoneki cepsep				Active bots	Server uptime	
-	Фильтры: ID:	IP:			Активных ботов:	Время работы сервера:	
	ID	Инфо		Комментарий		Проброшенные порты	
		Info		Comment		Forwarded ports	
ŀ							
ŀ							
ŀ							
l							
ŀ							
ŀ							
ŀ							
t							
l	Bot command		III	Comment			
1	Команда боту			Комментарий			
3	Быстрые команды	Launch Socks5	Port forwarding	Close all ports	Dirable bot	Kilobytes rece	ived
	Launch RDP Launc	h VNC				Принято килобайт	
	Запуск	3anyok 3anyok Socks5	Проброс	закрыть все	Отключить	Отправлено клиновант.	
			, nopia	порты		Сбросить логи	
						Peret logr	астройк

Figure 4: Translated C2 graphical user interface

These user interfaces included video management and playback applications as shown in Figure 5 and Figure 6 respectively. Tom will share some interesting work he did with these in a subsequent part of this blog series.

	Recorded video										
	Записанное видео					Add new nath	Delete current nath	De lete vider			*
ath to video	Путь к видео:				- До	бавить новый путь	Удалить текущий путь	Удаление видео			
	UIDs: Файлы:										
	IbUids		UID	IP бота	Имя файла	Название видео	Начало записи	Конец записи П	роцесс с которого пи	али	
				Bot IP	Filename	Video title	Start of recording	End of recording	Process from	which it was rec	orded
					Video	title	Header		Open by LIPI	Bug video	Open
Bots	Ботов:	Filter	Фильтр: UID	D:	Название	видео:	Заголовок:		Открыть по якорю	Баг видео	Открыть
		Handings	Заголовки:					-	Экспорт заголовков		Отмена
		neadings	Уникальных б	отов:	Видео:				Export headers		Cancel
		Unique bots			Video	-					

Figure 5: Translated video management application user interface

_	Player						
1	Проигрыватель	Bot	Header		Frames		
Ĩ	Файл:	Бот:	Заголовок:		▼ Кадры:	 Open vide o	Открыть видео
						Recorded video	Записанное видео
						Open by URL	Открыть по якорю
	<						
	Frame number	Total frames	Video length длина видео:	- 💠 🍸 Ti Bp	те мя:		Processed fran
	Размер кадра: Frame size	Длина кадра Frame leng	ı (байт): gth (bytes)	On server Ha	сервере: 10:17:42 F 🛫	Scale	Обработано кадров

Figure 6: Translated video playback application user interface

Figure 7 shows the backdoor builder that was contained within the RAR archive of operator tools.

-	Builder v1.1			_
	■П Билдер v1.1		X	
Configuration file	Файл конфига: Sample edit box		Сохранить конфиг	Save config
Prefix	Префикс: Sample edit box			
Input file	Исходный файл: Sample edit box		Сохранить конфиг как	Save config as
Output file	Конечный файл: Sample edit box		Создать ехе	Create exe
	Admin settings Настройки админки		Отмена	Cancel
Admin host (address)	Хосты (адреса) админок:	_		
	1: Sample edit box			
	2: Sample edit box			
	3: Sample edit box			
Beacon interval	Период отстука в админку: Sample Mин. Min.			
Admin password	Пароль для админки: Sample edit box			
	Server settings Настройки сервера			
Server addresses	Адреса серверов (обязательно указывать порт - ip:port): 1: Sample edit box 2: Sample edit box 3: Sample edit box Работа с сервером в отдельном процессе (рекомендуется) Work with servers in a separ При старте не соединяться с сервером Do not connect with servers upon starting Через сколько минут бездействия отключаться от сервера (максимум 99999 мин): Sample Disconnect from server after minutes of inactivity (maximum 99999 min) Defenserum	ate process (red	commended)	
Public key				
	Additional options Дополнительные опции			
Autostart	Автозапуск Плагины с сервера Get plugins from servers Запуск сплойта Работа в одном процессе (не использовать sychost exi	0		
Singleton test	Проверка запуска копии Work in one process (don't use svchost.exe)			
	Резильтат создания билла			
	Sample edit box Build results		*	

Figure 7: Translated backdoor builder application user interface

The operator RAR archive also contained an operator's manual explaining the semantics of all the backdoor commands. Figure 8 shows the first few commands in this manual, both in Russian and English (translated).

1. команда video	 video command
формат команды: video ИмяВидео [ИмяПроцесса]	command format: video NameVideo [ProcessName]
если имя процесса не указано, то пишется со всего экрана	if the process name is not specified, then it is written from t
имя видео должно быть обязательно	the name of the video is required
команда video off отключает видео запись	the video off command disables the video recording
команда запоминается и будет вновь запущена после ребута	the command is remembered and will be restarted after reboot
2. команда download	2. download command
формат команды: download [user\$ИмяЮзера] {url имя плагина}	command format: download [user \$ UserName] {url name of the p
загружает файл по урл или имени плагина, сохраняет его во временн	loads the file by URL or plugin name, saves it in a temporary f
если указан аргумент user, то если бот запущен под системой, то ф	if the user argument is specified, then if the bot is running u
если указано ИмяЮзера, то файл будет запущен от имени указанного	if you specify the UserName, the file will be launched on behal
3. команда аттуу	3. ammyy command
без параметров: если плагин не был загружен, то загружает аттуу.р	without parameters: if the plugin was not loaded, it loads ammy
При инсталяции узнается ид для подключения и отсылается на сервер	When installing, the ID is recognized for connection and sent t
параметр install - заново загружает плагин и инсталирует его	parameter install - reloads the plugin and installs it
параметр del - уничтожает запущенный процесс аттуу и удаляет плаг	parameter del - kills the running ammyy process and removes the
параметр stop - останавливает амми	stop parameter - stops ammyy
параметр run - запускает амми	parameter run - starts ammyy
при инсталяции амми ставится как сервис, если это не удалось, то	when installing, ammyy is set up as a service; if it fails, it
4. команда update	4. update command
Осуществляет горячее или холодное обновление бота	Performs hot or cold bot updating
Формат команды: update [cold] имя плагина (обновления)	Command format: update [cold] plug-in name (updates)
Если не указан cold (горячее обновление), то загружается указанно	If cold is not specified (hot update), then the specified updat
в тоже время текущий бот завершает работу	At the same time, the current bot is shutting down
Если указан cold (холодное обновление), то загружается указанное	If cold is specified (cold update), the specified update is dow
обновление вступит в силу только после ребута	The update will take effect only after reboot

Figure 8: Operator manual (left: original Russian; right: translated to English)

Down the Rabbit Hole: When Having Source Code Does Not Help

In simpler backdoors, a single function evaluates the command ID received from the C2 server and dispatches control to the correct function to carry out the command. For example, a backdoor might ask its C2 server for a command and receive a response bearing the command ID 0x67. The dispatch function in the backdoor will check the command ID against several different values, including 0x67, which as an example might call a function to shovel a reverse shell to the C2 server. Figure 9 shows a control flow graph of such a function as viewed in IDA Pro. Each block of code checks against a command ID and either passes control to the appropriate command handling code, or moves on to check for the next command ID.



Figure 9: A control flow graph of a simple command handling function

In this regard, CARBANAK is an entirely different beast. It utilizes a Windows mechanism called <u>named pipes</u> as a means of communication and coordination across all the threads, processes, and plugins under the backdoor's control. When the CARBANAK tasking component receives a command, it forwards the command over a named pipe where it travels through several different functions that process the message, possibly writing it to one or more additional named pipes, until it arrives at its destination where the specified command is finally handled. Command handlers may even specify their own named pipe to request more data from the C2 server. When the C2 server returns the data, CARBANAK writes the result to this auxiliary named pipe and a callback function is triggered to handle the response data asynchronously. CARBANAK's named pipe-based tasking component is flexible enough to control both inherent command handlers and plugins. It also allows for

the possibility of a local client to dispatch commands to CARBANAK without the use of a network. In fact, not only did we write such a client to aid in analysis and testing, but such a client, named botcmd.exe, was also present in the source dump.

Tom's Perspective

Analyzing this command-handling mechanism within CARBANAK from a binary perspective was certainly challenging. It required maintaining tabs for many different views into the disassembly, and a sort of textual map of command ids and named pipe names to describe the journey of an inbound command through the various pipes and functions before arriving at its destination. Figure 10 shows the control flow graphs for seven of the named pipe message handling functions. While it was difficult to analyze this from a binary reverse engineering perspective, having compiled code combined with the features that a good disassembler such as IDA Pro provides made it less harrowing than Mike's experience. The binary perspective saved me from having to search across several source files and deal with ambiguous function names. The disassembler features allowed me to easily follow crossreferences for functions and global variables and to open multiple, related views into the code.



Figure 10: Control flow graphs for the named pipe message handling functions

Mike's Perspective

Having source code sounds like cheat-mode for malware analysis. Indeed, source code contains much information that is lost through the compilation and linking process. Even so, CARBANAK's tasking component (for handling commands sent by the C2 server) serves as a counter-example. Depending on the C2 protocol used and the command being processed, control flow may take divergent paths through different functions only to converge again

later and accomplish the same command. Analysis required bouncing around between almost 20 functions in 5 files, often backtracking to recover information about function pointers and parameters that were passed in from as many as 18 layers back. Analysis also entailed resolving matters of C++ class inheritance, scope ambiguity, overloaded functions, and control flow termination upon named pipe usage. The overall effect was that this was difficult to analyze, even in source code.

I only embarked on this top-to-bottom journey once, to search for any surprises. The effort gave me an appreciation for the baroque machinery the authors constructed either for the sake of obfuscation or flexibility. I felt like this was done at least in part to obscure relationships and hinder timely analysis.

Anti-Analysis Mechanisms in Source Code

CARBANAK's executable code is filled with logic that pushes hexadecimal numbers to the same function, followed by an indirect call against the returned value. This is easily recognizable as obfuscated function import resolution, wherein CARBANAK uses a simple string hash known as PJW (named after its author, P.J. Weinberger) to locate Windows API functions without disclosing their names. A Python implementation of the PJW hash is shown in Figure 11 for reference.

```
def pjw_hash(s):
  ctr = 0
  for i in range(len(s)):
    ctr = 0xffffffff & ((ctr << 4) + ord(s[i]))
    if ctr & 0xf0000000:
       ctr = (((ctr & 0xf0000000) >> 24) ^ ctr) & 0x0fffffff
  return ctr
```

Figure 11: PJW hash

This is used several hundred times in CARBANAK samples and impedes understanding of the malware's functionality. Fortunately, reversers can use the <u>flare-ida scripts</u> to annotate the obfuscated imports, as shown in Figure 12.

📕 🚄 🖼	
1000CEE7	
1000CEE7	
1000CEE7	; Attributes: bp-based frame
1000CEE7	
1000CEE7	sub_1000CEE7 proc near
1000CEE7	
1000CEE7	arg_0= dword ptr 8
1000CEE7	
1000CEE7	push ebp
1000CEE8	mov ebp, esp
1000CEEA	push [ebp+arg_0]
1000CEED	push 0E3685D1h ; shlwapi.dll!PathFindFileNameA
1000CEF2	push 3
1000CEF4	call resolve_pjw
1000CEF9	pop ecx
1000CEFA	pop ecx
1000CEFB	Call eax
1000CEFU	pop eop
1000CEFE	retn sub 10000557 opdo
1000CEFE	Sun_Lanares such
TODOCEFE	

Figure 12: Obfuscated import resolution annotated with FLARE's shellcode hash search

The CARBANAK authors achieved this obfuscated import resolution throughout their backdoor with relative ease using C preprocessor macros and a pre-compilation source code scanning step to calculate function hashes. Figure 13 shows the definition of the relevant API macro and associated machinery.



Figure 13: API macro for import resolution

The API macro allows the author to type API(SHLWAPI, PathFindFileNameA)(...) and have it replaced with GetApiAddrFunc(SHLWAPI, hashPathFindFileNameA)(...). SHLWAPI is a symbolic macro defined to be the constant 3, and hashPathFindFileNameA is the string hash value 0xE3685D1 as observed in the disassembly. But how was the hash defined?

The CARBANAK source code has a utility (unimaginatively named tool) that scans source code for invocations of the API macro to build a header file defining string hashes for all the Windows API function names encountered in the entire codebase. Figure 14 shows the source code for this utility along with its output file, api_funcs_hash.h.



Figure 14: Source code and output from string hash utility

When I reverse engineer obfuscated malware, I can't help but try to theorize about how authors implement their obfuscations. The CARBANAK source code gives another data point into how malware authors wield the powerful C preprocessor along with custom code scanning and code generation tools to obfuscate without imposing an undue burden on developers. This might provide future perspective in terms of what to expect from malware authors in the future and may help identify units of potential code reuse in future projects as well as rate their significance. It would be trivial to apply this to new projects, but with the source code being on VirusTotal, this level of code sharing may not represent shared authorship. Also, the source code is accessibly instructive in why malware would push an integer as well as a hash to resolve functions: because the integer is an index into an array of module handles that are opened in advance and associated with these pre-defined integers.

Conclusion

The CARBANAK source code is illustrative of how these malware authors addressed some of the practical concerns of obfuscation. Both the tasking code and the Windows API resolution system represent significant investments in throwing malware analysts off the scent of this backdoor. Check out <u>Part Two of this series</u> for a round-up of antivirus evasions, exploits, secrets, key material, authorship artifacts, and network-based indicators. <u>Part Three</u> and <u>Part Four</u> are available now as well!