ITG23 Crypters Highlight Cooperation Between Cybercriminal Groups

Malware May 19, 2022

By Charlotte Hammond co-authored by Ole Villadsen , Golo Mühr 25 min read

IBM Security X-Force researchers have continually analyzed the use of several crypters developed by the cybercriminal group ITG23, also known as Wizard Spider, DEV-0193, or simply the "Trickbot Group". The results of this research, along with evidence gained from the disclosure of internal ITG23 chat logs ("Contileaks"), provide new insight into the connections and cooperation between prominent cybercriminal groups whose attacks often lead to ransomware.

Crypters are applications designed to encrypt and obfuscate malware to evade analysis by antivirus scanners and malware analysts. Crypters generally operate by encrypting the pre-compiled malware payload and embedding it within a secondary binary, known as a stub, which contains code to decrypt and execute the malicious payload. The use of crypters allows malware developers to easily experiment with different methods of evading antivirus detection without having to make changes to the malware itself.

X-Force analyzed thirteen crypters that have all been used with malware built or operated by ITG23 internal teams or third-party distributors — including Trickbot, BazarLoader, Conti, and Colibri — as well as malware developed by other groups such as Emotet, IcedID, Qakbot, and MountLocker. The presence of one of these crypters on a file sample is a strong indication that its developer, distributer, or operator is either a part of ITG23 or has a partnership with the group.

X-Force found evidence that ITG23 by mid-2021 scaled up their efforts to crypt malware with the development of several new crypters and the construction of a Jenkins build server to automate the crypting of malware at scale. X-Force also observed the analyzed crypters used repeatedly by Emotet and IcedID malware samples, indicating ITG23 is also crypting malware for these groups. These findings add to a growing body of evidence indicating a close relationship between ITG23 and the threat actors behind the development and operation of IcedID and Emotet.

Additionally, X-Force uncovered that at least one ITG23 crypter has been used repeatedly since late February 2022 with the Qakbot banking trojan and at least once with the Gozi banking trojan likely delivered by the ITG23 distribution affiliate TA551 (tracked by X-Force as Hive0106). X-Force's analysis of these crypters has also uncovered a previously undisclosed relationship between the IcedID group and MountLocker ransomware-as-a-service (RaaS) operation.

A Tangled Web They Weave

ITG23's "Build Machine"

ITG23 is a cybercriminal gang known primarily for developing the Trickbot banking Trojan, which was first identified in 2016 and initially used to facilitate online banking fraud. The group since that time expanded its operations to develop and operate new malware such as BazarLoader and Anchor. ITG23 also adapted to the ransomware economy by using its payloads to gain a foothold in victim environments for ransomware attacks and developing and operating the Conti and Diavol RaaS operations. ITG23 is best thought of as a group of groups, not unlike a large corporation, who report to common "upper management" and share infrastructure and support functions, such as IT and human resources. One of these support groups within ITG23 is dedicated to developing crypters for use with the group's own malware operations as well as for several other groups.

ITG23 have been crypting their malware for several years, and crypters used by the group were regularly seen in use with malware such as Trickbot, Emotet, Cobalt Strike and Ryuk. However, the development of multiple new crypters

during the past year suggests a focused effort to scale up their crypting operation.

Evidence gained from several sources, including ContiLeaks, indicates that ITG23 has set up a Jenkins build server to automate the mass crypting of malware, also referred to as the "Build Machine". Jenkins is an open-source automation server designed to automate the building, testing, and deploying of software. The "Build Machine" was created in April 2021, coinciding with an increase in the use of crypters with malware developed by ITG23 and other groups.

Since that time, ITG23 crypters have been applied to:

- Malware used to gain a foothold in victim environments, such as Trickbot, BazarLoader, Sliver, IcedID, Emotet, Qakbot, and Gozi. We even identified ITG23 crypters with Colibri, a loader advertised on underground forums that was used to download Trickbot in fall of 2021, likely by an internal ITG23 distribution affiliate. Some of these malware families are built by ITG23, such as Trickbot and BazarLoader, and others are built by different groups, such as IcedID, Emotet, and Qakbot. ITG23 distribution affiliates have deployed Sliver, an open source, cross-platform adversary simulation and red team platform, probably to gain access for ITG23 internal red teams to conduct ransomware attacks.
- Cobalt Strike beacon samples downloaded during attacks commencing with the above malware and used by internal red teams or other affiliates when performing ransomware attacks.
- Ransomware such as Conti and MountLocker, also known as Xinglocker, AstroLocker, and Quantum, which are often deployed following an infection with the above tools and malware.

ITG23 has discontinued use of Trickbot and BazarLoader as of December 2021 and February 2022, respectively, but X-Force continues to observe the crypters leveraged by other malware, including IcedID, Emotet, Conti, Qakbot, and the adversary simulation software Cobalt Strike. One notable exception is the Anchor malware which although attributed to ITG23 does not tend to use the same crypters as the other malware mentioned throughout this report. The Anchor malware was commonly observed using a separate crypter, named ShellStarter, which has some code overlap with Anchor itself and was likely created by the same developer. The ShellStarter crypter was also regularly used with Cobalt Strike payloads, but otherwise did not seem to be used for general crypting operations. We are also currently analyzing Bumblebee malware samples which we have also linked to ITG23 to determine if they are using an ITG23 crypter.

ContiLeaks

In February 2022, a Ukrainian security researcher using the Twitter handle "ContiLeaks" revealed a wealth of information about ITG23 and its operations, including private conversations between its members. While these leaks appeared to concentrate on the Conti RaaS operation, they also show that it was part of the larger ITG23 "corporation" which also includes ITG23's crypting operation. These chats indicate that the head of this crypting operation uses the handle "Bentley", who manages a team of developers responsible for both developing the crypters and crypting malware for affiliates and partners. Bentley in turn regularly provides status reports to "Mango", a more senior manager within ITG23 who reports to the group's former leader "Stern." Other security researchers have also identified Bentley and his role managing the crypting team. Below is an example of a status update on malware crypting that Bentley would send on a regular basis to Mango.

```
Date: Aug 26, 2021 @ 11:08:21.000
From: bentley@q3mcco35auwcstmt.onion
To: mango@q3mcco35auwcstmt.onion
Message:
Проект лео - 13 криптов. Билд машина (Project leo - 13 crypts.  Build Machine)
БК (BK)
группа 15: 20 криптов, билд машина (group 15: 20 crypts, build machine)
группа 19: 5 крипта, билд машина (group 19: 5 crypts, build machine)
группа 20: 1 крипто, билд машина (group 20: 1 crypt, build machine)
Трик: (Trick)
4 длл: 2 сэм 2 невил (4 dll: 2 sam 2 nevil)
Тройка: (Troika:)
```

```
невил (nevil)
Шелкод: билд машина (Shellcode: Build Machine)
Кобальт: билд машина (Cobalt: Build Machine)
```

Chat logs from the ContiLeaks also provide details about the creation of the build machine. On April 15, 2021, Mango informed Stern that the build machine for the crypters would be ready by the end of April 2021.

```
Mango → Stern:  билд машина для крипторов будет готова к концу месяца, вчера уже
начали обкатывать ее но пока сыровата
(The build machine for cryptors will be ready by the end of the month, yesterday they
already started to run it in, but it’s still raw)
```

On June 7, 2021, Bentley provides an update to Stern on the status of the transition of work to the build machine.

```
Bentley → Stern: Дела - хорошо. Интересно и насыщено.
Все крипторы перешли из ручного труда в автоматическуй сборку через билд машину.
Теперь они занимаются актуализацией и чисткой стабов. А файлы я делаю на билд машине,
 проверяю и выдаю.
Если что-то билдится грязным - обращаюсь к криптору. Он чистит стаб. Снова проверяем и
выдаем.
Задачи:
1. Криптование файлов для Лео на билд машине.
2. Шелкод кобальт
3. Локеры
4. Коабальт exe и длл
5. dll трика
6. Обучаю и предоставлюя доступ другим членам команды к билд машине, чтобы они могли
сами собирать крипты.
7. Подготовака линков для нагрузки и тестирование exелей для netwalker, hash, cherry.

Everything is OK. Interesting and rich.
All cryptors have moved from manual labor to automatic assembly through the build
machine.
Now they are engaged in updating and cleaning stubs. And I make files on the build
machine, check and issue.
If something is being built dirty, I turn to cryptor. He cleans the stub. Check again
and release.
Tasks:
1. Crypting files for Leo on the build machine.
2. Cobalt shellcode
3. Lockers
4. Cobalt exe and dll
5. Trickbot dll
6. Educate and give other team members access to the build machine so that they can
collect the crypts themselves.
7. Preparing links for loading and testing excels for netwalker, hash, cherry.
```

Within the ContiLeaks, there are multiple references to the use of a Jenkins server for the Build Machine. In one such example, on January 17, 2022, two ITG23 developers “derekson” and “elon” discuss the Jenkins server. X-Force also uncovered Program Database (PDB) file paths used by ITG23 crypters that reference Jenkins (see below for more details).

```
Derekson → Elon: Привет. Почти закончил со вторым сервером. Скажи когда можно
подключить к дженкинсу для теста?
```

```
(Hello. Almost finished with the second server. Tell me when can I connect to jenkins
for a test?)
```

Throughout the leaked chats, there are multiple examples of Bentley delivering crypted malware samples to affiliates and partners such as Cherry, Netwalker, and Zeus. X-Force assesses that “zevs” (“zeus”) is affiliated with the prominent distribution group Hive0106 (aka TA551), which used the gtags ‘zev,’ ‘zem’ and ‘zvs’ during their Trickbot campaigns. Hive0106 is a prominent distribution affiliate with an established relationship with ITG23. Throughout the chats, “zeus” is alternatively translated as “зевса”, “зевсом”, “зевсу”, and “зевс” depending on the grammatical case.

For example, on Aug 10, 2021, Bentley sends the following request to Hof, a developer associated with Trickbot malware:

```
Bentley → Hof:  Доброе утро. Сделай, пожалуйста, zev4.dll и  zem1.dll для Зевса
(Good morning. Please make zev4.dll and zem1.dll for Zeus)
```

The following messages also indicate crypted samples were prepared for Zevs:

```
August 31, 2021:
Bentley → Zevs: Еще ответ: у нас есть опыт серийной выдачи криптов п БК* уже, один
заказчик берет партиями по 30-100 штук
(Another answer: we have experience in the serial issuance of crypts and BK* already,
one customer takes in batches of 30-100 pieces)

September 24, 2021:

Neo → Zevs: монт молчит, я крипты готовил 3 штуки к 8 по мск
(Mont is silent, I prepared 3 crypts by 8 Moscow time)
```

*We assess БК (BK) likely is a reference to BazarLoader based on analyzing multiple chat references to this acronym.

Emotet and IcedID: Longtime Pals

The use of ITG23 crypters with Emotet and IcedID malware is the latest evidence of a close relationship with these groups that has featured distributing each other’s malware and cooperating on malware development. Emotet first appeared in 2014 as a banking trojan and later emerged as a prominent downloader for other banking trojans, including IcedID, Qakbot, and Trickbot. IcedID, also known as Bokbot and often referred to by ITG23 as Anubis, is a banking trojan first discovered by X-Force in September 2017. Since that time IcedID — like many banking trojans — has evolved to include backdoor and data harvesting capabilities and is often used as a downloader for other malware, including Cobalt Strike and ransomware.

Emotet: ITG23 and the Emotet group have a history of seeding each other’s malware. ITG23 has used Emotet extensively to deliver Trickbot malware often leading to the notorious Emotet -> Trickbot -> Ryuk ransomware attack sequence. Following actions to disrupt Trickbot group operations in fall 2020, Emotet moved quickly to assist ITG23’s recovery by downloading Trickbot malware to infected machines. A year later, ITG23 returned the favor by seeding Emotet samples to facilitate Emotet’s return following the January 2021 international law enforcement operation against the group.

The presence of “Veron” aka “Mors” participating in conversations with ITG23 members in the leaked chats also points to ITG23’s close cooperation with Emotet. Historically, “mors” was a gtag used with Trickbot samples delivered by Emotet. Based on the conversations, Veron/Mors appears to be a liaison to ITG23 for Emotet related matters. Veron/Mors also seemed to work with the crypting team, and messages can be found from Bentley which discuss crypting files for Veron. Bentley sent the following messages to Veron and Stern between February and May 2021 possibly related to crypting Emotet samples for testing purposes before Emotet’s reappearance in November:

```
February 24, 2021:
Stern → Bentley: veron запустился? (Veron started?)
```

```
Bentley → Stern: Он начинает в марте. Работаем над криптами для него. наших криптора
(He starts in March. We're working over the crypters for him. Our crypters)

March 1, 2021:
Stern → Bentley: veron не начал еще? (Veron hasn't started yet?)
Bentley → Stern: Првиет. Еще не начанал. Сделали годеый крипт его длл. Ждем как даст
полную версию со всеми ньюансами
(Hi.  Not started yet.  Made a suitable crypter for his dll.  We're waiting for a full
version with all the nuances.)

May 5, 2021
Bentley → Veron: Можешь дать длл на крипт? Пока можем начать криптовать и готовить
стабы
(Can you give a dll for the crypt? For now, we can start to crypt and prepare stubs)
```

Messages between Veron and Stern in May 2021 seem to suggest that the return of Emotet may have been delayed due a need to rewrite parts of the code for security purposes.

```
May 18, 2021:
Stern → Veron: привет. когда стартуем?
(Hi, when are we starting?)
Veron → Stern: привет, я скажу когда точно, в ближайшее время уже, делаю чтобы не
взломали
(Hello, I'll tell you exactly when, in the near future already, I'm doing it so that
they don't get hacked)

May 24, 2021:
Veron → Stern: привет, сорри, что задерживаем, но надо переписать часть, я за
безопасность
напиши как будешь, если вопросы есть
(hello, sorry for the delay, but we need to rewrite part, I'm for security
let me know if you have any questions)
```

IcedID: The first evidence of ITG23's cooperation with the IcedID group appeared in May 2018 when security researchers observed IcedID downloading Trickbot malware. Several months later other researchers noted Trickbot returning the gesture and downloading an updated IcedID variant that incorporated features used with Trickbot samples, suggesting that the two groups also collaborated on development. In early 2019, other analysts observed IcedID using a custom Trickbot shareDLL module to download core Trickbot malware. These researchers a month later described a new Trickbot proxy module for man-in-the-middle (MITM) attacks against web browsers that was highly similar to the IcedID proxy module. A Trickbot module named anubisDll32 was also developed containing the IcedID core code. In November 2021, X-Force and other researchers observed multiple campaigns during which BazarLoader was used to download IcedID malware.

ITG23's leaked chats provide additional insight into ITG23's close relationship with IcedID, although the exact nature of this relationship remains unclear. On May 1, 2021, Stern congratulates "Leo" on his "cool bot IcedID" for gaining the attention of security researchers, revealing that Leo is likely affiliated with the IcedID group.

```
Stern → Leo: а твой крутой бот ICEDId
(and your cool ICEDId bot)
Stern → Leo: про него пишут исследователи
(researchers write about it)
Stern → Leo: что ты сейчас на первом месте
(that you're in the first place)
```

The leaked chats often refer to a "Project Leo", which we assess is a reference to IcedID. Bentley regularly provides Mango with updates on crypting related to "Project Leo" and in November 2021, Stern messaged the following

instruction to Bentley:

```
Stern → Bentley: "включи крипты лео
(turn on the crypts of Leo)"
```

IcedID and MountLocker Ransomware

X-Force uncovered evidence that ITG23 crypters were used with MountLocker (see below), a ransomware-as-a-service (RaaS) operation that has been active since July 2020. Since then, MountLocker has rebranded several times to other names including XingLocker, AstroLocker, and Quantum. This evidence — combined with code overlap between IcedID and MountLocker ransomware and the use of IcedID alongside MountLocker during multiple ransomware attacks — suggests that the IcedID group operates the MountLocker RaaS.

The following conversation between Stern and Bentley on May 6, 2021, provides additional evidence that Leo, who operates IcedID, also has some involvement with ransomware. Stern asks Bentley which 'lockers', aka ransomware, his team have been crypting, and Bentley responds that they have had binaries from Reshaev and from Leo. Reshaev is a developer/manager for the Conti ransomware.

```
Stern: как автобилды работают?
Bentley: Большая часть стабов уже работают. Выдаем локеры exe 32 64 длл 32 64 , кобу
32 64 как exe так и длл. Шелкоды в exe и длл. Простые длл, БК.
Stern: какие локеры
Bentley: от решаева в exe и от лео в длл
Stern: How's the autobuild working?
Bentley: Most of the stubs are already working.  We issue lockers exe 32 64 dll 32 64,
cobalt 32 64 as exe and dll. Shellcode in exe and dll.  Simple Dll, BK.
Stern: Which lockers?
Bentley: From Reshaev in exe and from Leo in Dll
```

Analysis of IcedID and MountLocker samples reveals areas of code overlap, particularly in the logging and decryption functions. Both IcedID and MountLocker generate extensive debug logs, which are formatted in an almost identical manner.

's'	seg0... 00000030 C	[INFO] bot.hooker.process > inject to [%u] %s\r\n
's'	seg0... 00000030 C	[INFO] bot.inj.config > set apc id=%u size=%u\r\n
's'	seg0... 00000037 C	[ERROR] bot.dg.cookie.chrome > sqlite exec status=%u\r\n
's'	seg0... 00000036 C	[INFO] bot.inj.config > config set ok id=%u crc=%u\r\n
's'	seg0... 000000... C	[INFO] bot.inj.replace.text > replaced=%s\r\n
's'	seg0... 00000024 C	[INFO] bot.bc.socks > new sock=%p\r\n
's'	seg0... 00000006 C	[INFO]
's'	seg0... 00000033 C	[ERROR] bot.shed > ITrigger_QueryInterface=%0.8X\r\n
's'	seg0... 00000024 C	[ERROR] bot.hooker.inject > write\r\n
's'	seg0... 00000032 C	[ERROR] bot.hooker.inject > open process gle=%u\r\n
's'	seg0... 000000... C	[ERROR] bot.gate.queue.add > merge/pack\r\n
's'	seg0... 00000027 C	[INFO] bot.url.get > item=%u list=%u\r\n
's'	seg0... 0000001F C	[INFO] bot.inj.traf > url=%s\r\n
's'	seg0... 00000037 C	[ERROR] bot.dg.pass.chrome > sqlite exec_2 status=%u\r\n
's'	seg0... 000000... C	[INFO] bot.dg.pass > ie vault status=%u\r\n
's'	seg0... 00000035 C	[ERROR] bot.dg.pass.chrome > sqlite open status=%u\r\n
's'	seg0... 000000... C	[ERROR] bot.inj.config > query apc gle=%u\r\n
's'	seg0... 000000... C	[INFO] bot.cmd > run shellcode param=%s\r\n
's'	seg0... 00000042 C	[WARN] bot.update.botpack > Bad format support=%0.2X file=%0.2X\r\n
's'	seg0... 0000002F C	[INFO] bot.dg.pass > cred status=%u count=%u\r\n
's'	seg0... 00000037 C	[ERROR] bot.dg.pass.chrome > sqlite exec_1 status=%u\r\n
's'	seg0... 000000... C	[ERROR] bot.bc.main.session > auth=%0.8X\r\n
's'	seg0... 00000028 C	[INFO] bot.bc.socks > sock=%p host=%s\r\n

Figure 1 — Debug log strings from an IcedID sample

```
ext "UTF-16LE", '[ERROR] locker.file > set_pos gle=%u name=%s',0Dh,0Ah
ext "UTF-16LE", 0
lign 10h
                        ; DATA XREF: sub_18000499C+141↑o
ext "UTF-16LE", '[ERROR] locker.file > read gle=%u name=%s',0Dh,0Ah,0
lign 10h
                        ; DATA XREF: sub_18000499C+12B↑o
ext "UTF-16LE", '[ERROR] locker.file > write gle=%u name=%s',0Dh,0Ah
ext "UTF-16LE", 0
lign 10h
                        ; DATA XREF: sub_180004B10+48↑o
ext "UTF-16LE", '[ERROR] locker.file > open gle=%u name=%s',0Dh,0Ah,0
lign 10h
                        ; DATA XREF: sub_180004B10+88↑o
ext "UTF-16LE", '[ERROR] locker.file > get_size gle=%u name=%s',0Dh,0Ah
ext "UTF-16LE", 0
                        ; DATA XREF: zf_encrypt_file+110↑o
ext "UTF-16LE", '[ERROR] locker.file > rename gle=%u name=%s',0Dh,0Ah
ext "UTF-16LE", 0
lign 10h
                        ; DATA XREF: zf_encrypt_file+186↑o
ext "UTF-16LE", '[ERROR] locker.file > write_key gle=%u name=%s',0Dh
ext "UTF-16LE", 0Ah,0
lign 20h
                        ; DATA XREF: zf_encrypt_file+231↑o
ext "UTF-16LE", '[OK] locker.file > time=%0.3f size=%0.3f KB speed=%'
ext "UTF-16LE", '0.3f MB/s name=%s',0Dh,0Ah,0
lign 10h
                        ; DATA XREF: zf_encrypt_file+265↑o
ext "UTF-16LE", '[OK] locker.file > time=%0.3f size=%0.3f MB speed=%'
ext "UTF-16LE", '0.3f MB/s name=%s',0Dh,0Ah,0
lign 20h
                        ; DATA XREF: zf_handle_net_drive+2E↑o
ext "UTF-16LE", '[SKIP] locker.work.enum.net_drive > readonly name=%'
ext "UTF-16LE", 's',0Dh,0Ah,0
lign 10h
```

Figure 2 — Debug log strings from a MountLocker sample

Additionally, samples of both IcedID and MountLocker were identified which contained almost identical XOR decryption and key generation algorithms.

```
1  __int64 __fastcall zf_decrypt_data(unsigned int *a1, __int64 a2)
2  {
3    unsigned __int16 i; // [rsp+20h] [rbp-18h]
4    unsigned __int16 v4; // [rsp+24h] [rbp-14h]
5    unsigned int v5; // [rsp+28h] [rbp-10h]
6    __int64 v6; // [rsp+40h] [rbp+8h]
7
8    v5 = *a1;
9    v4 = *(a1 + 2) ^ *a1;
10   v6 = a1 + 6;
11   for ( i = 0; i < v4; ++i )
12   {
13     v5 = zf_gen_key(v5);
14     *(a2 + i) = v5 ^ *(v6 + i);
15   }
16   return a2;
17 }
```

```
1  __int64 __fastcall zf_gen_key(int a1)
2  {
3    return __ROL4__(__ROL4__(__ROR4__(__ROR4__(__ROR4__(a1 + 11865, 1), 1), 2) ^ 0x151D, 2), 1);
4  }
```

Figure 3 — XOR algorithm and key generation function from an IcedID sample

```
__int64 __fastcall zf_decrypt_data(unsigned int *a1, __int64 a2)
{
  unsigned __int16 i; // [rsp+20h] [rbp-18h]
  unsigned __int16 v4; // [rsp+24h] [rbp-14h]
  int v5; // [rsp+28h] [rbp-10h]
  char *v6; // [rsp+40h] [rbp+8h]

  v5 = *a1;
  v4 = *(a1 + 2) ^ *a1;
  v6 = a1 + 6;
  for ( i = 0; i < v4; ++i )
  {
    v5 = zf_gen_key(v5);
    *(a2 + i) = v5 ^ v6[i];
  }
  return a2;
}
```

```
__int64 __fastcall zf_gen_key(int a1)
{
  return (-__ROR4__(__ROL4__((a1 ^ 0x93FE) + 30784, 1) - 23205, 1) - 116602);
}
```

Figure 4 — XOR algorithm and key generation function from a MountLocker sample

Qakbot: A New Partner?

While monitoring for signs of ITG23 crypters' use in the wild, X-Force identified the first known use in late February 2022 of an ITG23 crypter with Qakbot aka Qbot. The Qakbot banking trojan was first identified in 2007 and like other banking trojan groups, it has increased its functionality over the years and evolved into a flexible downloader and backdoor often leading to ransomware attacks. The appearance of ITG23 crypters on Qakbot samples provides evidence of a direct relationship between ITG23 and the Qakbot group. The relationship between ITG23 and Qakbot is also supported by additional evidence published recently. That said, the discovery does not come as a complete surprise. In the leaked chats, "Tramp" asked Bentley on December 6, 2021, about crypting Qakbot:

```
Tramp →  Bentley: криптанем квак бота ?
(crypt Quak Bot?)
Bentley →  Tramp: давай попробуем
(let's try)
```

Tramp later sends Bentley a file named stager_1_tr.dll to be crypted. Tramp may be affiliated with "TR", a prominent distribution affiliate also known as TA577 and which is currently distributing Qakbot. We have since identified ITG23 crypters used with Qakbot samples delivered by the two most prominent and current Qakbot distribution affiliates — TA570 and TA577 — suggesting that ITG23 is assisting the Qakbot group with crypting its malware and not just a single distribution affiliate. There is also evidence that Qakbot has a relationship with the Emotet group, dating back several years. Emotet has historically been used to download Qakbot in addition to Trickbot, for example during 2020 and then more recently in March 2022. Given ITG23's partnership with Emotet, it is possible that the Emotet group is facilitating ITG23's relationship with Qakbot leading to the latter's use of an ITG23 crypter.

Hive0106 (TA551) Gozi Sample

X-Force researchers also found a Gozi sample using an ITG23 crypter on April 7, 2022 (see below). Gozi is also a banking trojan first appearing in 2007 that has evolved into a multi-module, multi-purpose malware. However, unlike the other banking trojans discussed so far, the Gozi source code has leaked and the malware is not operated or developed by a single group. The threat actor Hive0106 (aka TA551) was likely responsible for this campaign delivering Gozi. We assess that Bentley and his team likely crypted this Gozi sample on behalf of this group, with which they have an established relationship.

The Crypters

Crypters are applications designed to encrypt and obfuscate malware to protect it from anti-virus scanners and malware analysts. The crypting process generally involves encrypting a pre-compiled malware payload, such as an

EXE, DLL file, or shellcode, and embedding it within a secondary binary, known as a ‘stub’, which contains code to decrypt and execute the malicious payload. The stubs generally take the form of binaries, such as Exe or DLL files, are often either polymorphic or updated frequently in order to evade signature-based detection methods, and usually make use of code obfuscation techniques.

When the crypted binary is executed, the stub code will extract the embedded payload, decrypt it, load it into memory and execute it. As a result of this behavior, the crypted binary containing the stub code may also be referred to as a ‘loader’ or ‘in-memory dropper’.

In order to protect their payloads many crypters may also include additional functionality to detect sandbox environments, hinder AV scanners, escalate privileges, or perform other basic system checks. It’s common for crypters to utilize a high level of code obfuscation within the stubs, and the majority also employ polymorphic techniques such as metaprogramming to ensure that each crypted binary is unique and thus make it harder to identify via signature-based detection methods.

Another common technique is for the crypter to disguise the malware as a benign executable, and to this end, they will often use source code from legitimate applications as a template for the stub binary, or include strings or functions which mimic benign activity. The code to decrypt and load the payload will be made as inconspicuous as possible in an effort to hide it from the attention of malware scanners.

All of these techniques together also provide obstacles for the malware reverse engineer and make it harder to write detection signatures and automated malware parsers.

X-Force research indicates that ITG23 is providing crypting services to other threat actors in addition to using them for their own malware. Using the same crypter for multiple malware families has an additional benefit of confusing the identification capabilities of AV applications. Indeed, it is not uncommon to see a crypted malware binary flagged by AV as belonging to one malware family, when it is in fact a completely different one, and they just happen to be using the same crypter.

X-Force analysts are tracking at least thirteen crypters we believe to be developed and currently in use by ITG23 that we are calling Dave, Pear, Lore, Mirror, Galore, Rustic, Tron, Hexa, Stub, Error, Skeleton, Charm, and Graven. Whilst variants of the Dave RC4 crypter have been in use for at least a couple of years, the rest appear to have been primarily developed in the past year. ITG23 has used these crypters with Trickbot, BazarLoader and Conti malware — all of which are attributed to ITG23 — and used them to crypt malware on behalf of groups such as IcedID and Emotet. We have also observed these crypters used with Cobalt Strike samples, which we assess are used by ransomware internal red teams or affiliates when conducting attacks on clients infected with Trickbot, BazarLoader, IcedID or Emotet.

In the sections below, we provide an overview of each of the crypters and the examples of the malware families they have been used with.

Dave

Dave is one of the older crypters that X-Force tracks as currently in use by ITG23, having been used since at least 2020. Several variations of Dave exist, but one of the most common variants stores the payload either as an RCData type resource or within the data section, and decrypts it using a custom RC4 algorithm, which uses a variable sbox size rather than the standard for RC4 which is 256 bytes. Dave is so-called as it commonly wraps the payload in a second-stage shellcode loader, where the ascii string ‘dave’ is used to mark the end of the payload. Dave loaders have been most frequently observed loading Emotet and Trickbot, but also occasionally BazarLoader, Ryuk, Conti, IcedID, Cobalt Strike and Colibri.

It is common practice for malware developers to ‘strip’ malware binaries during compilation which removes symbol information such as variable and function names. This has the benefits of making the malware more difficult to analyze, as well as removing details, which may potentially be used by analysts to fingerprint the developer.

Almost all samples analyzed by X-Force are fully stripped, however from November 2021 to January 2022 X-Force observed a number of unstripped Dave-crypted samples uploaded to repositories such as VirusTotal, providing a rare

insight into the coding style of the developer. Based off some of the strings and function names X-Forced determined the developer utilized components of publicly available code for the stub, for example, the function **CLoad::FromMemory()** can be traced back to a 2016 code sample, memlib.cpp, originally published on a forum. The aforementioned shellcode with the 'dave' signature also appears to be modified from the open source sRDI repository.

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  int (__stdcall *v4)(int, _DWORD, int, int, int, _DWORD); // esi
  int v5; // ebx
  int v6; // eax
  int (__stdcall *v7)(_DWORD, int, int, int); // esi
  int v8; // ebx
  int v9; // eax
  unsigned int v10; // [esp+8h] [ebp-40h]
  unsigned __int8 *v11; // [esp+28h] [ebp-20h]
  unsigned __int8 *module_handle; // [esp+30h] [ebp-18h]
  CLoad *v13; // [esp+3Ch] [ebp-Ch]

  hmod_current = hinstDLL;
  reson_call = fdwReason;
  reserv = lpvReserved;
  if ( _Z6AntiAVv() )
  {
    printf(&Format);
    return 0;
  }
  else
  {
    q1 = 0;
    q2 = 0;
    q3 = 0;
    q4 = 0;
    q5 = 0;
    module_handle = get_module_handle(L"kernel32.dll");
    get_module_handle(L"ntdll.dll");
    virt_allocation_mem = get_proc_address(module_handle, 0x49E979A2u);
    virtmemNuma = get_proc_address(module_handle, 0x309911E5u);
    if ( virtmemNuma )
    {
      v4 = virtmemNuma;
      v5 = atoi("64");
      v6 = atoi("8192");
      BYTE1(v6) |= 0x10u;
      v13 = v4(-1, 0, 143872, v6, v5, 0);
    }
    else
    {
      v7 = virt_allocation_mem;
      v8 = atoi("64");
      v9 = atoi("8192");
      BYTE1(v9) |= 0x10u;
      v13 = v7(0, 143872, v9, v8);
    }
    memcpy(v13, &rawData, 0x23200u);
    v11 = malloc(size_arrary);
    rc4_init(v11, key, 0x61u);
    rc4_crypt(v11, v13, 0x23200u);
    hLibrary = CLoad::LoadFromMemory(v13, 0x23200, v10);
    return 1;
  }
}
```

Figure 5 — Unstripped Dave stub with original function and variable names as assigned by the developer.

Select samples using the Dave crypter:

Sample Family	SHA256 Hash
Cobalt Strike	a9c4eafcff0567c68919c93ddf8baa769392e92706e6b35f7b989310d70f732f
Colibri	f5fd02ebd2376fd1bc1ff121e9bfda618755a5c049edc8a4288eb67eb1cc7f9b
Emotet	5da102cc1ff7d842e3b5c9d6f571bd3b3afdc1715d37f120b31e1859928f5837
Trickbot	947c81aefdb479de7e75f14be2921bb829478680e039c2bc40a4c258524819b8
BazarLoader	47bac27be954cf593ac731cd57fa98b565cf5036a6fbf35c508549f039eea8f3
Conti	5ace33358a8b11ae52050d02d2d6705f04bd47a27c6c6e28ef65028bbfaf5da9
Ryuk	180f82bbedb03dc29328e32e054069870a1e65078b78b2120a84c96aaed7d843

Pear

Pear crypter can be tracked back to at least March 2021, when it was used to crypt IcedID. Pear has been primarily observed in use with IcedID payloads, but samples loading BazarLoader, Trickbot, and Colibri payloads have also been found. Pear crypter stores the payload within one of the stub binary's data sections, and custom algorithms are used to encrypt the payload. The exact format and values of the encryption algorithm change per sample, suggesting a technique such as metaprogramming may have been employed to generate the algorithms. The encrypted payload often has a recognizable alternating byte pattern that makes use of a restricted set of bytes in order to keep the entropy low. Entropy measures the level of randomness in the data, and many encryption algorithms will generate encrypted data with a distinctively high entropy value, which is easily detectable by binary analysis tools. By using an algorithm that outputs lower-entropy data, the encrypted payload is less easy to detect by automated systems.

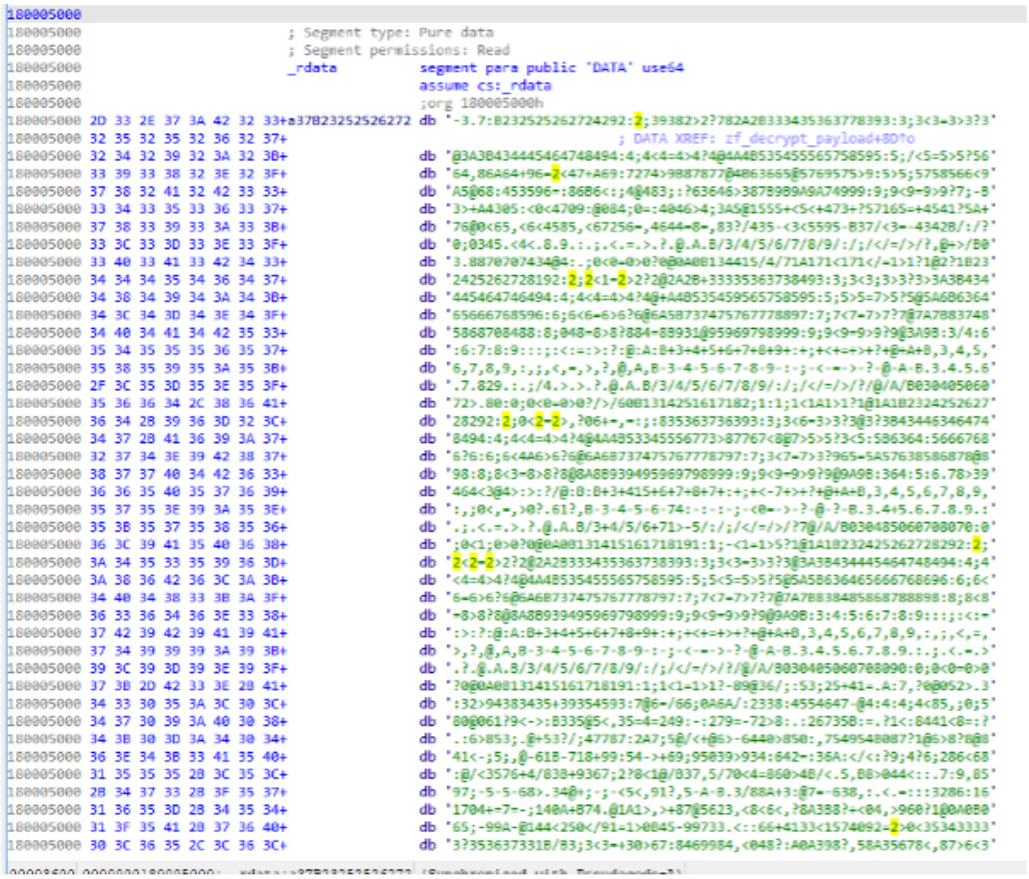


Figure 6 — Pear crypted sample with distinctive encrypted payload utilizing a restricted byte set.

Select samples using the Pear crypter:

Sample Family	SHA256 Hash
IcedID	9f4bdbfec9f091e985e153a1597fc271abd0320c60dfe37dc3e7d81e5d18ad83
BazarLoader	26cac671e215d88b5070af7d94200588d2b7c414a6e8debf7370b993fcfffb23
Colibri	b1fc2855f5579f02ac6d03c2d20e85948e9609fd769389addb8ce5986b1f8ecd
Trickbot	e2ba0567ac236a24bfd4df321ae7860e8fe2810dbd088e0e90d67167c1ccd4c5

Lore

Lore crypter has been in use since at least May 2021 and has been observed with payloads including Emotet, Trickbot, BazarLoader, IcedID and Cobalt Strike. This crypter stores the payload as a BITMAP type resource, with a 103-byte bitmap file header added to the start of the payload data. Upon execution, the stub code loads the resource, removes the bitmap header, and decrypts the remaining data using XOR and a hardcoded key. The payload is then loaded into memory and executed. The crypter originally appeared to be designed for use with PE payloads, and so shellcode-based payloads were wrapped in an additional second stage loader.

Lore crypted binaries often include a lot of extraneous imports and junk functions in an attempt to obscure the location of the payload decryption and loading code from analysts. This loading code instead uses API hashes to

retrieve handles to the API functions it requires, so the extraneous imports can generally be ignored by the analyst.

A handful of Lore crypted samples were identified containing the following PDB paths:

```
204506c69824371017f482e88f9fbb14cfd0fbc17233fa8d3ffbf4f527e20af5
 c:\jenkins\workspace\crypter5_generic_exe\Bin\x64\Release\MFC_Stub.pdb
d1a12e52d9fcc57580146370933a3f9eb027c5fec972abc9ac2f2b7d9f94e0d3
 c:\jenkins\workspace\crypter5_shellcode_64_exe\Bin\x64\Release\MFC_Stub.pdb
41c56e92efd01a553d0faf39ccb440c7e84d32531335c262572d6a01bf7f70c8
 c:\jenkins\workspace\crypter5_generic_exe\Bin\x86\Release\MFC_Stub.pdb
615f9a5517e71648a0780c186af8642e2848589d6962bc12ff34c0c54b650df5
 c:\jenkins\workspace\crypter5_shellcode_64_exe\Bin\x64\Release\MFC_Stub.pdb
```

These paths provide evidence of a Jenkins server being used for crypting operations and also suggest that it likely contains a number of different crypters, with crypter5 being Lore Crypter. This is corroborated by the PDB path found within some Error crypted samples, detailed further below, which refer to it as ‘crypter7’.

The directory names ‘crypter5_generic_exe’ and ‘crypter5_shellcode_64_exe’ indicate that different configurations of the crypter stubs were likely compiled for different types of payloads. In this case, the two samples containing the reference ‘crypter5_shellcode_64_exe’ are both 64-bit executable files that contain Cobalt Strike shellcode http stagers as their payloads. For the two samples containing the reference ‘crypter5_generic_exe’, one is a 64-bit executable containing a BazarLoader payload and the other is a 32-bit executable containing a Conti ransomware executable.

Select samples using the Lore crypter:

Sample Family	SHA256 Hash
Cobalt Strike	ee8efcd34db429697337d7275d713385600c510558a8a4615bd1eb18847f43f2
Conti	e6e248be24782f28a492055ebb35886ad057d8a5ff4d7315f22af1fe29d9df0d
IcedID	7a6c42343b3d422c9f6f5c72763645b8f1b4931c609c320e60816aee55e4ae8a
Emotet	70b66e57ea54f48a8b288d65d93063478e27b5710cab106cf41464e086e784db
Trickbot	2587e94f3bc1ae54ff7732984925def76de934b3e1b1f7407bd66491db18f7e0
BazarLoader	8661bd7d893fe1dd2109fac55cf9cea5f609012769732039e20165a3198c1086

Mirror

Mirror crypter has been observed since November 2021, and so far has primarily been found loading BazarLoader payloads, as well as some IcedID and Cobalt Strike. Mirror crypter shares some code overlap and obfuscation mechanisms with Lore crypter, suggesting they may have the same developer or codebase. Mirror splits its encrypted payload into three parts which are stored across different sections of the resulting binary loader. Two main variants of the Mirror stub code have been found so far, one which decrypts the payload using AES-256 via the Windows CryptDecrypt API, and a second which decrypts the payload using XOR and a hardcoded key.

Select samples using the Mirror crypter:

Sample Family	SHA256 Hash
BazarLoader (IDB Sample)	cbd830c745bbec26733214798fe144c61ef4bac342c853f8a08b682077b2178b
BazarLoader (XOR variant)	b44d0261823595b303bdae62df7790b30c13a0a897978d30f3041c27a645eac6
IcedID	00c46232cdad873bf02787746fba9d196a6045bac1051154af7772f5b0f29b87
Cobalt Strike	9eedbac3f1c8795cf1f04301ecf2d66aacacbbb9e6c087ed158f00f81fae7375

Galore

Whilst the majority of ITG23’s current crypter stubs are coded in C/C++, it seems the developers also experimented with alternative languages, producing crypters with loader stubs written in both the Go and Rust programming

languages. Galore crypter uses the Go programming language and has been observed in the wild since mid-2021 when it was frequently used to crypt BazarLoader payloads. The Go programming language has become increasingly popular with malware authors over the past few years due to its convenient cross-platform support, and the fact that it produces large and complex binaries upon compilation which can be tricky to reverse engineer and often have lower detection rates against AV applications than their C/C++ coded counterparts.

Upon execution the Galore stub code decrypts the payload using XOR, and loads and executes the PE payload using code based off the open-source Reflective DLL Injection project. The use of this Reflective Dll Injection code is common in many of ITG23's crypters.

Select samples using the Galore crypter:

Sample Family	SHA256 Hash
Baazarloader	26be0ba3533703f5eeea8489e6a8881461dab7f597f33e546182ba1910953d09

Rustic

Rustic crypter uses the Rust programming language which, like Go, has been seeing an increase in popularity with malware developers. The payload is stored in the .rdata section of the loader and encrypted using a XOR based algorithm with two keys applied in multiple iterations. The crypter supports both shellcode and PE payloads, with shellcode payloads loaded into memory and executed directly, and PE payloads loaded in a similar manner to Galore crypter, using the Reflective DLL Injection technique.

Rustic crypted samples were first observed in early September 2021 and it has been used with malware including BazarLoader, IcedID, Cobalt Strike, Quantum, as well as implants from Sliver which is a post-exploitation framework written in Go.

```
19  v14 = -2i64;
20  sub_180019AC0();
21  *key2 = xmmword_1800283C0;
22  *&key2[16] = 538353430;
23  *&key2[20] = -19630;
24  *key1 = 0xFC9D778EEE985F27ui64;
25  *&key1[8] = -923177876;
26  *&key1[12] = -10794;
27  v0 = hHeap;
28  if ( !hHeap )
29  {
30    ProcessHeap = GetProcessHeap();
31    if ( !ProcessHeap )
32      goto LABEL_14;
33    v0 = ProcessHeap;
34    hHeap = ProcessHeap;
35  }
36  count = 0;
37  buffer = HeapAlloc(v0, 0, 0x9C3AD4ui64);
38  if ( !buffer )
39LABEL_14:
40    sub_180026EF0(10238676i64, 1i64);
41  buffer_ = buffer;
42  zf_copy_bytes(buffer, g_payload, 0x9C3AD4ui64);
43  do
44  {
45    for ( i = 0i64; i != 10238676; i += 2i64 )
46    {
47      v6 = &key1[-14 * (i / 0xE)];
48      buffer_[i] ^= v6[i];
49      buffer_[i + 1] ^= v6[i + 1];
50    }
51    ++count;
52  }
53  while ( count != 1001 );
54  for ( j = 0; j != 1001; ++j )
55  {
56    for ( k = 0i64; k != 10238676; k += 2i64 )
57    {
58      v9 = &key2[-22 * (k / 0x16)];
59      buffer_[k] ^= v9[k];
60      buffer_[k + 1] ^= v9[k + 1];
61    }
62  }
63  lpMem = buffer_;
64  payload_start = VirtualAlloc(0i64, 0x9C3AD4ui64, 0x3000u, 0x40u);
65  zf_copy_bytes(payload_start, lpMem, 0x9C3AD4ui64);
66  payload_start();
67  HeapFree(hHeap, 0, lpMem);
68}
00000592 LoadBytes:66 (180001192) (Synchronized with IDA View-A)
```

Figure 7 — Rustic stub loader code responsible for loading and decrypting the payload

```
00000077  C  .C:sovpxwrdnpcrsgtqklsvhexggdpyibirjgfxcrnodcuaifqzaafuzgbipfgyxbvzlfoieoeodxyzdlvphiydxjsve...
00000017  C  fatal runtime error: \n
0000003D  C  called `Option::unwrap()` on a `None` valuestack backtrace:\n
00000058  C  note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.\n
00000036  C  __rust_begin_short_backtrace__rust_end_short_backtrace
0000000A  C  <unknown>
00000088  C  C:gtjokngjkqidefqnfptxsxsmywrmdvsbccizvzizewydcwmyfkxkljtkpojvfwnldxbemcqtrxgecfbcquchpi...
00000080  C  C:ypqksajtzgthifjjcfnszeyvmmjchxtkyvqdagffxwbnzsyaoigexcbcrzoeskxzlkswtfkfdzxtsaemfyqjlvcoq...
0000002C  C  called `Result::unwrap()` on an `Err` value
00000024  C  memory allocation of  bytes failed\n
0000001D  C  Rust panics must be rethrown
00000086  C  C:pebtfsdvphiyapkgswmsvgdwxuagxizwocntokmlzdflmbosoeddfjbibsybgonesmwnqsyawmquttw...
00000024  C  <unnamed>thread '' panicked at '',
0000004F  C  note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace\n
```

Figure 8 — Strings within a Rustic-crypted sample indicate that the binary was written using the Rust language

Select samples using the Rustic crypter:

Sample Family	SHA256 Hash
Sliver	45aa8efb6b1a9a0e0091040bb99a7c37d346aaf306fa4e31e9d5d9f0fef56676
Cobalt Strike	e75fce425df2e878c7938cdf86c8e4bde541c68f75d55edb62a670af52521740

BazarLoader	8d84152b69161bf5abb2f80fef310ec92cc8b1cb23dff18eebd8d039cda8f8ad
IcedID	bec6dc7f7bfbded59d1a9290105e13ac91cf676ef5a4513bacbfcabf73630202
Quantum	fd7ca7af9b2b6c5ffdb3206d647301de8bea33a69679e117be30e9a601c5dea2

Tron

Tron crypter first appeared in the wild in September 2021 when it was used to crypt Trickbot binaries associated with gtag rob132. Since then, it has been observed with payloads within Emotet, Trickbot, BazarLoader, IcedID, Conti and Cobalt Strike. Of note, Tron is the crypter identified in this article from CERT-UA.

Tron crypted binaries have their payload usually stored within the .text section of the stub loader which, upon execution, unpacks and decompresses the payload, and then loads it into memory and executes it. The decompression of the payload is performed using the Zlib library; however, the unpacking appears to be performed using code originating from an obscure Github project called Megatron (https://github.com/akakist/megatron/), specifically a module called ioBuffer.cpp which implements basic buffer manipulation and unpacking functions. The Megatron project has since been taken down but previously strings from the source code in Github could be observed within the unpacking functions in the crypted binaries.

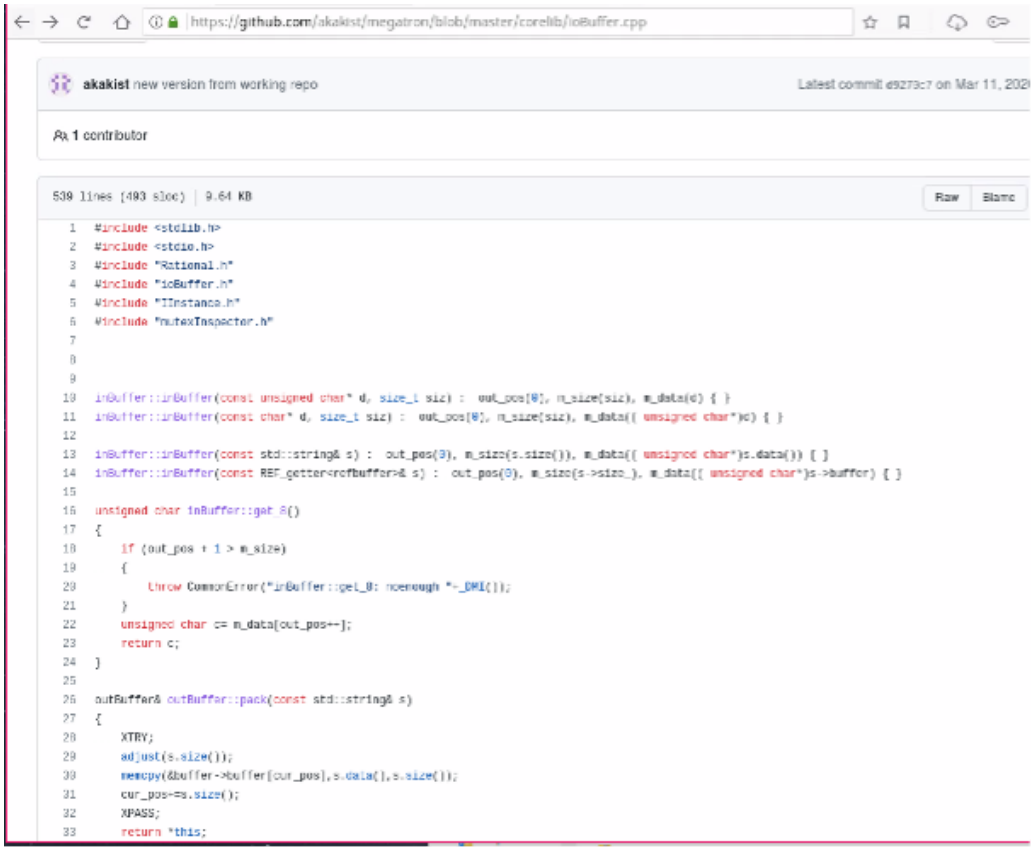
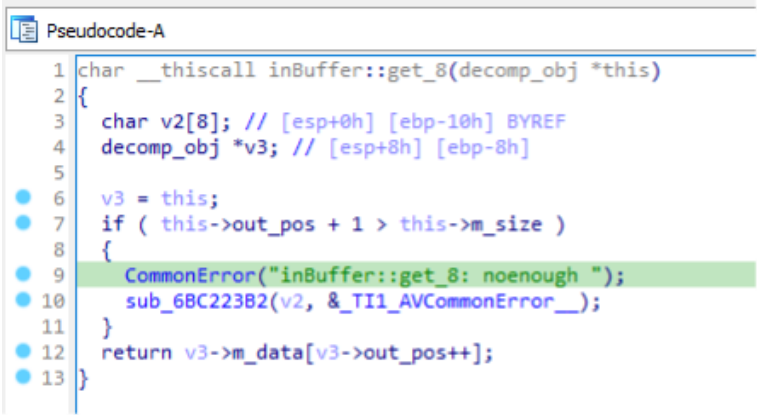


Figure 9 — The source code of ioBuffer.cpp as seen on Github

The above image shows the source code of ioBuffer.cpp as seen on Github, specifically a function named **inBuffer::get_8()** is shown, which contains the error string "**inBuffer::get_8: noenough**". This same function and error string can be seen within the unpacking functions of the crypted binary.



The payload data is split into chunks which are delimited with the bytes ‘c3 cc cc cc’, where the number of ‘cc’ bytes varies based on alignment. Bytes used to calculate the size of each chunk are added at the start of each chunk. The unpacking code parses the payload data, calculating the size of each chunk and appending the chunk data to the output buffer whilst checking for and discarding the 0xc3 and 0xcc padding bytes.

The compressed and decompressed sizes are then parsed from the start of the unpacked data, and the zlib.decompress function is used to decompress the payload. One version of this crypter stores the payload in multiple parts, which are unpacked individually and then joined together before decompression.

Several other variants of the Tron crypter have also been observed. One example contains the same ioBuffer unpacking functions, but the payloads are decrypted using XOR rather than decompressed using Zlib. Some variants also have the payload stored in the .data section, and others may encode the payload in a numeric ascii format.

Some samples were identified containing path strings for header files such as the following:

```
Z:\cr4\ballast\5\core\src\BitArray.h
Z:\cr\crypter4\ballast\3\openjp2\opj_intmath.h
```

Considering the PDB strings identified within Lore and Error crypted samples, these path strings may indicate that Tron crypter is referred to as crypter4 within the group.

Select samples using the Tron crypter:

Sample Family	SHA256 Hash
Cobalt Strike	44e2057c7466881a61e3b542ce055b3d54aa7d88040ce879a915e20ed996d097
Conti	38784c635de9716c09a6f11f4d76f6402b5f6638f1614ed929c7de136bb5301a
Emotet	8d8138c23bf514a984918f7b5c5a7501e91b2c058574b7ce0b9ccbe638e82387
Trickbot	fd083bc2dbc3426a332eaf861dea03c648ad04cb73ba8f09504c970af9134898
BazarLoader	b88382ef06808155253f631a06e31024436e19d5bffd34f9b03906295e82de52
IcedID	2b9cba43290c9d4cc2d6a47432ddac5752c63e5ac519c2056ba466580424ed3b

Hexa

Hexa crypter compresses and RC4 encrypts its payload, and then encodes it as a hexadecimal ascii string to reduce entropy. This is then stored in the data sections of the stub binary, with some variants splitting the payload across two or three different sections. Upon execution the payload is reconstructed, decompressed and decrypted and then copied to a newly created memory section and execution transferred to the payload. Portable executable (PE) payloads may be preceded by a shellcode loader which is responsible for properly mapping the PE file into memory and executing it.

Hexa makes use of code obfuscation techniques to hinder analysis efforts including splitting the code into many tiny blocks separated by jumps, and the inclusion of blocks of junk data.

Hexa crypted samples were observed towards the end of 2021, with payloads including BazarLoader, Cobalt Strike, and Conti. It has also seen an increase in usage over the past couple of months where it has been used with malware families including IcedID, QakBot and Gozi.

Select samples using the Hexa crypter:

Sample Family	SHA256 Hash
IcedID	bbefa9f7747822e017580206931aec6e948e6cb3ca897b9615d87430b99e7d1e
Qakbot	0da8df441dc92d6719092aea1d3e9709e802aa87410279374d69626573fd3177
BazarLoader	18bbaddacba7bcdda4a1a088a640e167271f44d6232c20aa7d88eceeb3028826
Cobalt Strike	b51465ca7e71da2cd29072c819076c4efccb391dea353f16a36b0a60459b3358
Conti	c77032c772e0ef0e3200edf38223f9c6047e56294e840ea79689b9e56048c69c
Gozi	41ae907a2bb73794bb2cff40b429e62305847a3e1a95f188b596f1cf925c4547

Stub

Stub crypter was first observed in November 2021. It has been used primarily in IcedID campaigns, but samples have also been identified with payloads such as BazarLoader, Cobalt Strike, Conti, and Quantum ransomware, which is a variant of MountLocker and thought to be associated with the IcedID group.

Stub crypter stores the payload across multiple RCDATA type resources with sequential ids, e.g. 200, 201, 202. The first resource contains the encryption key, and the remaining resources each hold an encrypted section of the payload PE file.

To generate the encryption key, the malware takes the first resource, removes a 62-byte header, and then proceeds to generate each byte of the key by combining the next three bytes from the resource data using bitwise shift and or operations. The final key length is usually 1024 bytes.
The malware then proceeds to decrypt the next resource using this key and a custom xor-based algorithm, which varies between samples. The first decrypted resource contains the PE header of the payload binary, and the loading code uses information from this header to map each of the remaining PE sections into memory as it decrypts them from the resources. The loaded payload is then executed at its entry point.

Select samples using the Stub crypter:

Sample Family	SHA256 Hash
BazarLoader	936426ce7210fbd0ce519fb4121289fc1c43247fa96a7d1cd96d276f1662df26
Quantum	faf49653a0f057ed09a75c4dfc01e4d8e6fef203d0102a5947a73db80be0db1d
Cobalt Strike	84f1e4c2524fea85c43f9df6ac1449c95d2d3ba5bd7cb6bff2f4e1c97dc8cbe1
IcedID	008a674e33435ce0b892d0a68ac6d01f9606c040da87b21a10ed069729ee04ff
Conti	41896f40197a6160fcab046b5fc63a36d0805dbb1ca5a03af35b92b27d9a0eb5

Error

Error crypter was prominent from late November 2021 to January 2022 when it was used to crypt samples in Emotet, IcedID and BazarLoader campaigns, as well as being used with Cobalt Strike payloads. Error crypted binaries contain a large amount of junk code and strings for obfuscation, with one variant seemingly designed to be disguised as a hospital administration tool. Some samples also contain strings, which appear to have been generated from literary texts such as ‘David Copperfield’.

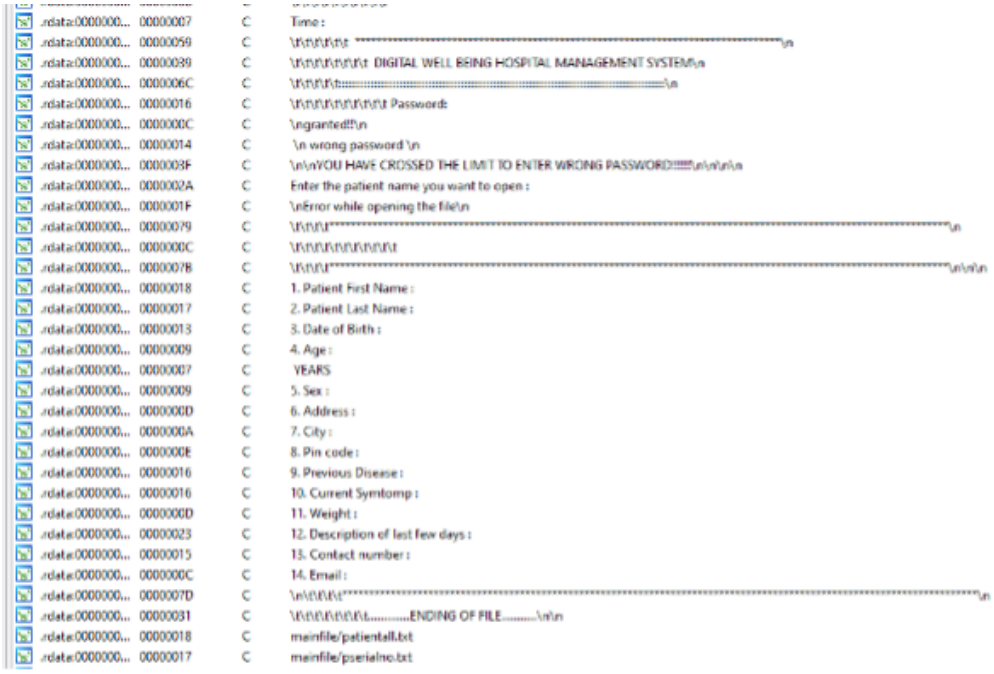


Figure 10 — Error crypted sample containing strings relating to a hospital administration application

Error crypter encrypts its payload using XOR and the encrypted payload is divided into small chunks which are scrambled up and stored. Upon execution, the stub code uses a complicated series of functions to retrieve the data chunks and reconstruct the encrypted payload. The XOR key required for decryption is generated in a similarly convoluted manner, with data being decrypted and retrieved from various sources and concatenated to form a string. This string is then hashed, and the hash is used to generate the final XOR key.

An example of one of the strings used to generate the XOR decryption key is as follows:

```
2021-12-03-
mok.35022336.17:33:40===700524802745472.xKUzpAWUHQuKEHhnAwJ4MEDN4oDSNpNqXpt.2691200820897.
```

Error crypter also includes some anti-debugging functions within the stub code including checking for the presence of a debugger and checking the system time year against a hard coded value.

Some Error crypted samples were found to contain the following PDB string:

```
C:\\crypter7\\Bin\\x64\\Release\\Dll\\cryptERRDll.pdb
```

This PDB string suggests that this crypter may have been known as ‘crypter7’ or ‘cryptERR’ internally within ITG23.

Select samples using the Error crypter:

Sample Family	SHA256 Hash
Emotet	a7343086d72f81f91cedc05d88b11cf44ba5da9ac6c25983870f3a77f854f4e9
BazarLoader	f17718d8f12cfada48a9288bf5f91e81787e361071f82345364c8e85b539524a
Cobalt Strike	1d20191aee650fd8c58c6564ce9ff5b86138a954bc49a3e25033cc888fc85466
IcedID	f9f62722ff249e8219d4864dc46a1bbb3871b1b3f9c4139ffe2726b8f6f27ad0

Charm

Charm crypter was observed primarily in campaigns between August 2021 and October 2021, and has been seen loading payloads such as BazarLoader, Cobalt Strike, Conti, and MountLocker. Charm crypter compresses its payload using an arithmetic coding algorithm, and then xor encrypts the compressed data and splits it into many small segments which are stored throughout the loader binary. Charm crypted binaries are obfuscated using junk code to hinder analysis.

Select samples using the Charm crypter:

Sample Family	SHA256 Hash
BazarLoader	8758196b4266ca7809e54c84ff6767784cb105fce247ad3459a15bb8ef9032c8
Cobalt Strike	6eccc2f0b5fb42a7b59881acdef621cc086d6ab76dfd80e5a3b3542590197805
Conti	63061a372c41f5797f18dfeed166ec350e4029c46ad3c42ff79b8e284eb65ad6
Quantum	267f6ba1363b2dbf56ad7e324380782de682a59f7d647eaee7d92b1ba5d2fcfa

Graven

Graven crypter splits the payload into three parts which are stored in different sections of the generated loader binary. Each part is then split into small pseudo-randomly sized chunks, delimited with pseudo-randomly sized chunks of null bytes. The algorithm to determine both the size of payload chunks and null-byte chunks is deterministic with a fixed seed allowing for the payload to be reconstructed by the loader. Upon execution, the payload is rebuilt and decrypted using AES, then loaded into memory and executed. Some variants of Graven also include code to create a mutex with the name 7ce3e80173264ea19b05306b865eadf9.

Graven crypted samples were primarily observed between November 2021 and February 2022, and payloads include BazarLoader, Emotet, and IcedID.

Select samples using the Graven crypter:

Sample Family	SHA256 Hash
BazarLoader	4246dbf6daf37bac0e525bdd8122131bedf4e32f9542c4696fa525e1f71a6508
Emotet	836d8e2f36ad80f937a377f568d78653e975e4b52db995ae18272dfecca9ac0f
IcedID	a61b1d70d469b8ca7acdbd26fc859e6aeb229c4636fe9c92eac856914f326ac8

Skeleton

Skeleton is a fairly basic crypter, which stores the payload as a XOR encrypted, MessageTable type resource within the loader binary, often with just a hardcoded ascii string used as the XOR key. Upon execution, the payload resource is loaded, decrypted, and executed in memory. Variants have been found loading either shellcode or PE formatted payloads. PE payloads are mapped into memory, imports loaded, and then executed from their entrypoint. Skeleton crypted binaries have been observed loading Trickbot, Cobalt Strike and IcedID payloads between December 2021 and late March 2022.

Select samples using the Skeleton crypter:

Sample Family	SHA256 Hash
Trickbot	01c69d0acc8734993ba9cbfe9b0da4616bb05041e103afdb487759995b93ee5c
IcedID	617e0f57f4283ca044003326663b5614d66f97e16bccdd8bec1321fad44a7195
Cobalt Strike	3dea0bac5c9ae010b4abeb532a3a347cd55682512ffe287dbb310d5d434777ef