CYBERARK®

REPORT

# BEHIND THE HIDDEN CONVERSION OF ELECTRICITY TO MONEY:

An In-Depth Analysis of XMR Cryptominer Malware

www.cyberark.com

## Abstract

2017 saw a major shift in the threat landscape from cyber crime monetization tactics, such as ransomware and credit card theft, to a new breed of malware called cryptominers, which take over a computer's resources and use them for illicit cryptocurrency mining. Fast-forward to 2018 and the use of malicious cryptominers has surged as cyber criminals rush to exploit a new cryptocurrency, based on the concepts behind Bitcoin, called "Monero." Several successful multi-million dollar heists tied to this open source, peer-to-peer cryptocurrency were reported in the first few months of 2018 alone.

The ability to easily convert electricity and access to computer hardware into money is something new in the world of cyber crime. As it significantly lowers the bar to entry for less-skilled attackers, criminal activity has skyrocketed. In stark contrast to ransomware, malicious cryptominers can run silently in the background, without any user interaction, indefinitely.

This has incentivized sophisticated attackers to hone their methods, making their malware as silent and low-impact as possible, so as to maximize the duration of their mining operations on compromised systems for maximum profit. This confluence of factors creates a new and formidable set of challenges for organizations and security vendors alike.

This paper provides an in-depth look at current cryptominer trends and the technology behind Monero. Additionally, it explores the projected attack methods that threat actors are likely to adopt, as well as tactics and best practices that security teams can utilize to keep themselves on the right side of this emerging cat-and-mouse game.

# Table of Contents

## Introduction

Monero (XMR) is an open source, peer-to-peer cryptocurrency with an emphasis on anonymity and decentralization. Also known as a "privacy coin," Monero emerged in April 2014, following in the footsteps of Bitcoin and aiming to improve upon its technology.

It uses a blockchain system, secured by a proof-of-work (PoW) algorithm, which relies on computational power (aka "mining") to validate new transactions and propagate them throughout its network. The demand for computational power to sustain the Monero blockchain exploded in 2017 and began to attract attention from a group of individuals who have access to some of the greatest computing power in the world: malware writers.

Millions of computers both at home and in organizations, infected with malware and unwitting botnet members, suddenly offered the opportunity of a lifetime to criminals who could use them to mine XMR. Malware monetization techniques such as ransomware began to fall to the wayside, and executable programs became silent vampires, siphoning electricity and converting it into cash for hackers all over the world. At the time of this writing, the Monero market value is approximately $3 billion.

This paper aims to provide insight into why Monero is game changing in this new paradigm, and why it remains the cryptocurrency of choice for hackers. It examines how malicious Monero mining works in practice, and some of the trends surrounding cryptomining we are likely to see—from both an offensive and defensive perspective—in the security space going forward.

## CryptoNight

### The Incentive

While the anonymity of Monero (and the fungibility it presents) is a valuable feature to malware authors, it is not the primary attraction. Monero's adoption and popularity has surged amongst malicious cryptominers because of the algorithm—CryptoNight—that's used to implement its PoW blockchain system.

CryptoNight has specific qualities that make it perfect for malicious cryptomining. As a result, a wave of innovative new Monero mining botnet malware has surfaced, resulting in anonymous profits of hundreds of thousands to millions of dollars in the past year alone. Notably, one of the latest commercial botnet malware families netted approximately $7 million in just six months.

With the advent of multi-factor authentication, more sophisticated security software and law enforcement tactics (including advanced blockchain analytics that effectively de-anonymized Bitcoin), and the widespread adoption of security chips on credit cards (eliminating the traditional botnet monetization tactic of cloning stolen credit cards), it seemed that the commercial botnet economy would steadily decline, despite the rise of ransomware in 2016 to 2017. Monero and currencies like it have breathed new life into a declining underground, minimizing risk for attackers while offering astronomical monetary gains to even amateur cyber criminals.

A recent Fortinet article cited an interesting statistic: "According to a Coinhive FAQ, they (the malware writers) chose Monero (XMR) because the algorithm used to compute the hashes is heavy, but better suitable to CPU limits, especially when compared to other cryptocurrencies where using GPUs (graphical processing units) would make a big huge difference. They mentioned that the benefit of using a GPU for Monero is about 2x, where it's 10,000x for BitCoin/Ethereum."

This in conjunction with the anonymity and tremendous potential profits involved in illegal Monero mining, is it any wonder malware authors are flocking to it in droves and abandoning old methods of botnet monetization, such as ransomware, spam and credit card theft?

According to Symantec's 2018 annual threat report, "The growth in coin mining in the final months of 2017 was immense. Overall coin-mining activity increased by 34,000 percent over the course of the year, while file-based detections of coinminers on endpoint machines increased by 8,500 percent. There were more than 8 million coin-mining events blocked by Symantec in December 2017 alone." Figure 1 illustrates this based on statistics collected by Symantec.

Furthermore, "Coinminers made up 24 percent of all web attacks blocked in December 2017, and 16 percent of web attacks blocked in the last three months of 2017." This is a good illustration of the shifting trend toward cryptomining as the monetization technique of choice for malware writers.

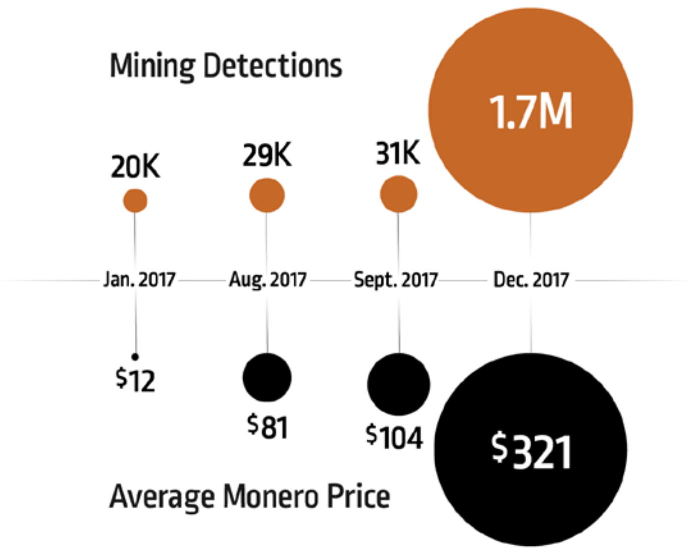


*Figure 1. Mining detections have increased exponentially throughout 2017 according to statistics from Symantec.*

## Diluting Satoshi's Vision

In the CryptoNote whitepaper, the CryptoNight PoW algorithm is described as having a primary goal "to close the gap between CPU (majority) and GPU/FPGA/ASIC (minority) miners." It goes on to say, "It is appropriate that some users can have a certain advantage over others, but their investments should grow at least linearly with the power. More generally, producing special-purpose devices has to be as less profitable as possible."

The reason for this is to stay true to the initial concept behind Satoshi Nakamoto's (the original author of the Bitcoin technology) Bitcoin whitepaper that proposed a concept of "one CPU, one vote" when finding P2P consensus for its blockchain. In the early days of Bitcoin, PoW (using SHA-256) was achieved in a highly decentralized fashion amongst the Bitcoin community by running full Bitcoin nodes on their personal computers using CPU mining for transaction block validation (i.e., mining). However, with the advent of GPU mining through CUDA and OpenCL, this old decentralized network structure was replaced with highly centralized clusters of high-power graphics cards, eventually culminating in the advent of ASIC miners designed specifically to calculate SHA-256 hashes using GPUs.

## Leveling the Playing Field

The authors of Monero intended to utilize CryptoNight to seize back network control from centralized mining operations. By intentionally taking the emphasis away from sheer computing power (where GPU and ASIC dominance were inevitable), they placed it instead on other factors that would disadvantage GPUs and once again make CPU mining feasible. They achieved this by using a memory-hard algorithm that forced memory latency into the forefront as a bottleneck for high-intensity computational power.

Specifically, they achieved this by elaborating on a concept put forth by the first Bitcoin fork (called Litecoin) in its PoW algorithm (Scrypt), wherein a block of pre-initialized, pseudo-random scratchpad memory is required to compute hashes. The concept was first introduced by Martin Adabi, who stated such functions were those "whose computation time is dominated by the time spent accessing memory." More specifically, what is happening is that a large block of pseudo-random data is allocated in memory, and the values it contains (which are accessed in an intentionally unpredictable way by the algorithm) are required to produce the hashed output.

The end result of such algorithms is a leveling of the playing field between GPU and CPU units. According to the CryptoNote whitepaper, "GPUs may run hundreds of concurrent instances, but they are limited in other ways: GDDR5 memory is slower than the CPU L3 cache and remarkable for its bandwidth, not random access speed." In this way, Satoshi's vision of "one CPU, one vote" is once again revitalized and the average home computer once again has a part to play in the P2P consensus network for the blockchain.

Despite the high emphasis on anti-ASIC mechanisms within the PoW algorithm itself, some ASIC manufacturers have managed to create devices that give substantial advantages to their owners over the average CPU. As of April 6, 2018, the core developer team behind Monero upgraded the PoW algorithm to CryptoNightV7 in response to the revelation that an ASIC manufacturer named Bitmain had created a

CryptoNight ASIC called Antminer X3 and was selling it on the open market. As a result of this PoW update, the Monero network hashrate (the combined power of all miners on the Monero network) dropped nearly 80 percent, indicating that Bitmain had been using the Antminer X3 to mine Monero in secret leading up to its announcement of the sale to the general public. This is illustrated by Figure 2, which shows the sharp decline in XMR network hashrate as of the anti-ASIC upgrade.

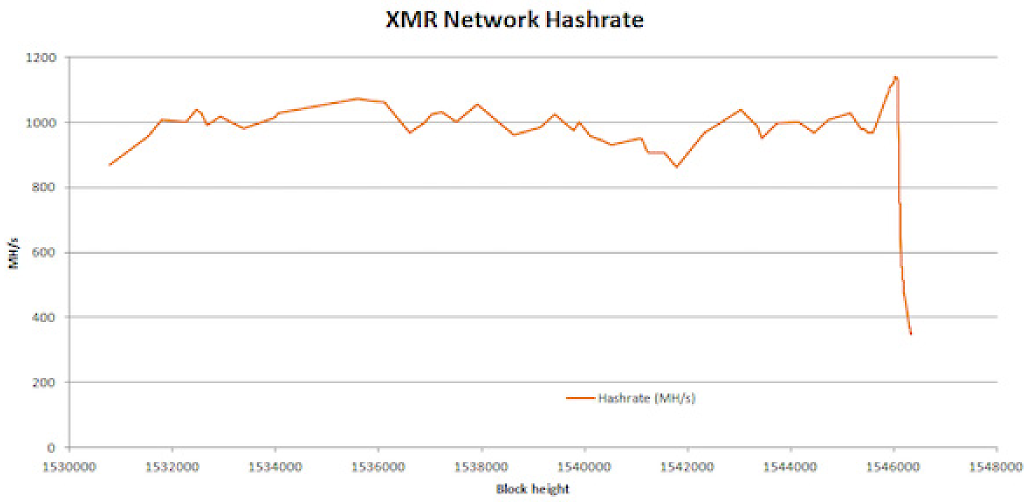


*Figure 2*

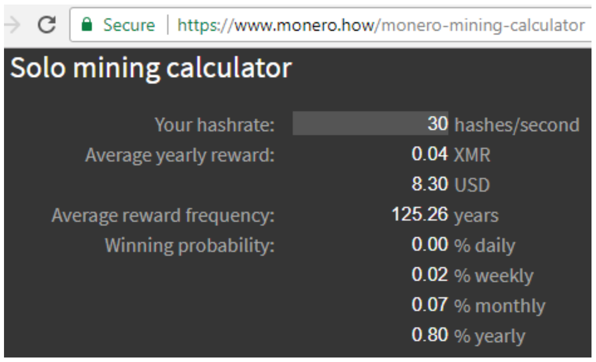
This incident clearly illustrates that in addition to thwarting ASICs through memory latency hard PoW algorithms, it is essential that a cryptocurrency continuously update its PoW algorithm so as to render ASICs periodically obsolete. The team behind Monero is currently one of the only teams in the cryptocurrency space that takes their anti-ASIC commitment seriously, and while it does preserve Satoshi's vision of "one CPU, one vote," it also preserves Monero as an ideal target for malicious cryptominers.

## Economics of the ASIC Farm Versus the Botnet

On the surface, the rewards of running a Monero mining botnet seem quite low. With a mid-range consumer Intel® Core™ i5-6300U CPU @ 2.40GHz generating about 30h/s (thirty CryptoNight hashes per second) at an average price of $210/XMR since early 2018, the average payout would be $8.40/year for solo mining, according to this Monero mining calculator.

This average is calculated based on the reality of how cryptocurrency mining actually works, wherein a miner receives absolutely no compensation unless they are the lucky one to solve the PoW algorithm (awarding them approximately 4.95 XMR on average since early 2018, worth $1,039.50 at $210/XMR). At a rate of 30h/s, the average CPU miner has a statistical likelihood of solving a single block only once every 125.26 years. Figure 3 illustrates this through a public solo mining calculator available on monero.how.

It's important to keep in mind that this average does not take into account electricity costs, which are considerable in a legitimate commercial mining operation to such an extent that mining is often unprofitable without a source of low-cost electricity. This is why so many ASIC cryptomining operations have relocated to countries such as Iceland.



Solo mining calculator

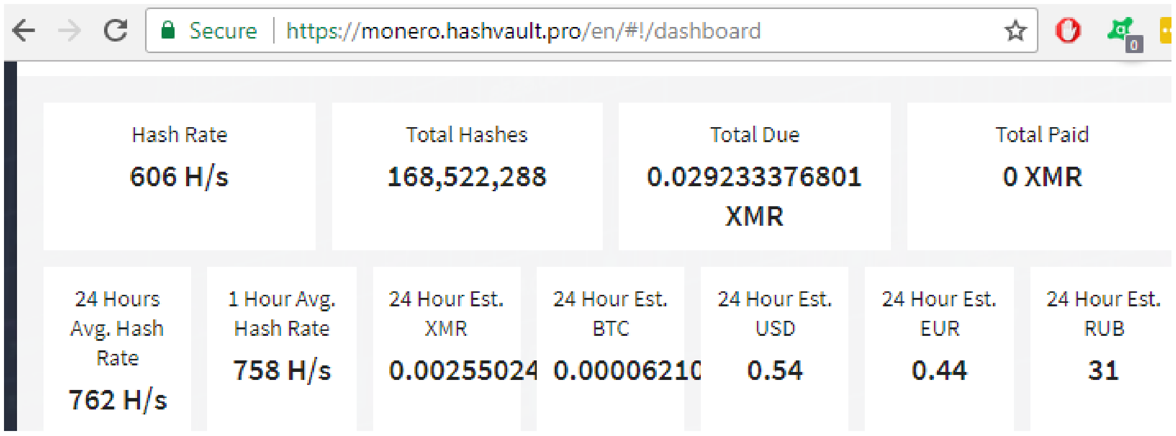
| | |
|---|---|
| Your hashrate: | 30 hashes/second |
| Average yearly reward: | 0.04 XMR |
| | 8.30 USD |
| Average reward frequency: | 125.26 years |
| Winning probability: | 0.00 % daily |
| | 0.02 % weekly |
| | 0.07 % monthly |
| | 0.80 % yearly |

*Figure 3*

The reality of the profitability of Monero botnets is actually quite different due to the phenomenon of pool mining. Monero, like many other cryptocurrencies, has reached a stage in its development where solo mining is no longer practical due to the astronomically low odds of ever actually solving a block and being paid a reward. For this reason, thousands of individuals typically pool their resources together (all using their hardware to guess the correct PoW answer) on the same transaction block in order to have a fighting chance of actually solving a block and getting paid. Once the mining reward (in this case, approximately 4.95 XMR) is given, it is split between all of the different individuals who contributed to the PoW based on the amount of computational power they each provided.

This provides consistent and reliable profits rather than one-off jackpots, and allows even the average home computer to generate a small and consistent revenue by contributing to a mining pool.

To demonstrate this, we mined on several of our own machines (which have above average CPU power), achieving a total hash rate of around 760h/s. The following dashboard from the pool mining website HashVault is helpful in illustrating the profit model (See Figure 4 ).


Secure | https://monero.hashvault.pro/en/#!/dashboard

| Hash Rate | Total Hashes | Total Due | Total Paid |
|---|---|---|---|
| 606 H/s | 168,522,288 | 0.029233376801 XMR | 0 XMR |

| 24 Hours Avg. Hash Rate | 1 Hour Avg. Hash Rate | 24 Hour Est. XMR | 24 Hour Est. BTC | 24 Hour Est. USD | 24 Hour Est. EUR | 24 Hour Est. RUB |
|---|---|---|---|---|---|---|
| 762 H/s | 758 H/s | 0.00255024 | 0.0000621C | 0.54 | 0.44 | 31 |

*Figure 4*

With an average payout of approximately $0.54 worth of XMR per day, there would be an average payout of $.00071 per H/s per day. With this data in mind, consider an example of a small botnet owner with control over 50,000 machines who CPU-mined Monero at an average rate of 30h/s. This would equate to an average hash rate of 1,500,000 h/s, which, on a daily basis, would translate to an average of $1,065 per day, or $31,950 per month. On the other side of the spectrum, major commercial botnets comprise hundreds of thousands or even millions of machines, and maximize profits by employing techniques to substantially increase hashrates by optimizing memory access via large page virtual memory in Windows and other methods. Figure 5 illustrates the process of malicious Monero mining from an attacker's perspective.

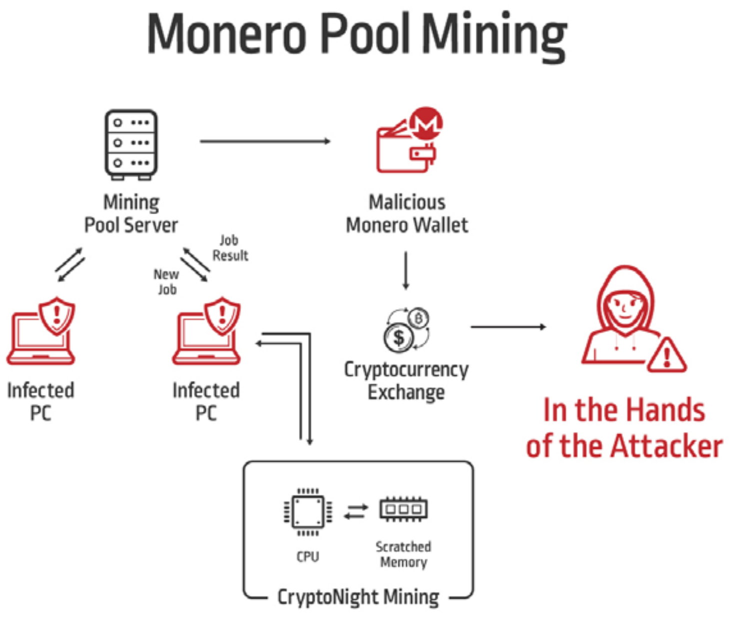
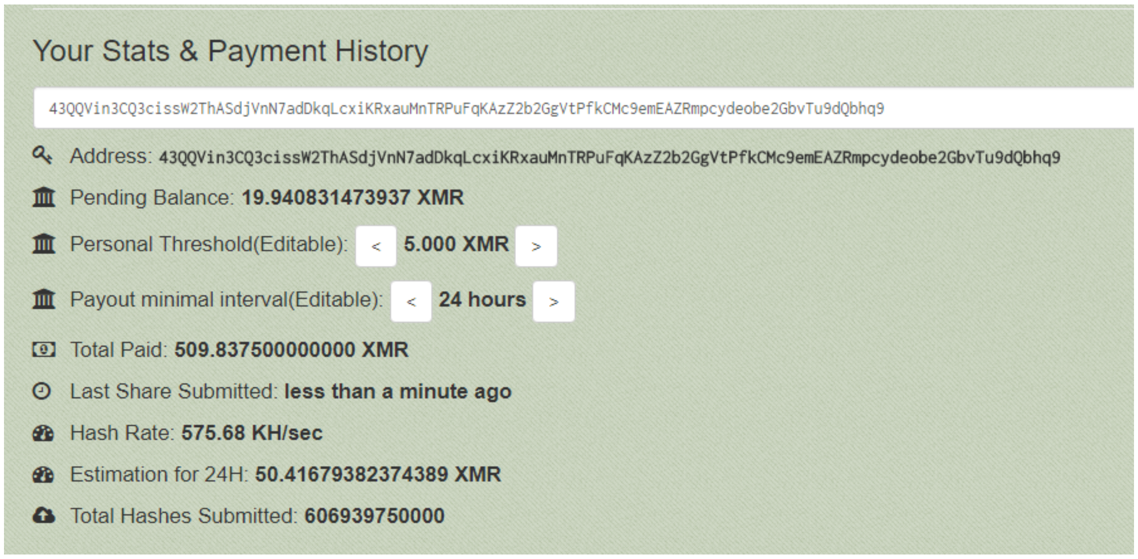


*Figure 5*

These huge returns on investment are consistent with the Adylkuzz worm, which was able to net hundreds of thousands of dollars' worth of Monero (millions at the current Monero price) in a matter of months by targeting Windows servers vulnerable to the EternalBlue exploit with above average CPUs. A prior analysis of this worm was able to query payout statistics from one of the mining pools it was using. See Figure 6 for details.



Your Stats & Payment History

43QQVin3CQ3cissW2ThASdjVnN7adDkqLcxiKRxauMnTRPuFqKAzZ2b2GgVtPfkCMc9emEAZRmpcydeobe2GbvTu9dQbhq9

Address: **43QQVin3CQ3cissW2ThASdjVnN7adDkqLcxiKRxauMnTRPuFqKAzZ2b2GgVtPfkCMc9emEAZRmpcydeobe2GbvTu9dQbhq9**

Pending Balance: **19.940831473937 XMR**

Personal Threshold(Editable): < **5.000 XMR** >

Payout minimal interval(Editable): < **24 hours** >

Total Paid: **509.837500000000 XMR**

Last Share Submitted: **less than a minute ago**

Hash Rate: **575.68 KH/sec**

Estimation for 24H: **50.41679382374389 XMR**

Total Hashes Submitted: **606939750000**

*Figure 6*

With a payout of about 50 XMR per day, this single payout address (and there were many others according to the analysis) was netting about $1,350 per day at a price of about $27/XMR at the time. At today's prices, this would translate to a payout of $10,500 per day, with a total payout of over $100,000 to this payout address alone.

The trend started by Adylkuzz continued more recently with Monero miner worms spreading through EternalBlue such as Smominru, which makes an estimated average of $8,400 per week at current Monero prices/mining difficulty level (which has increased substantially since the original Adylkuzz outbreak).

## Darknet Adoption

There is evidence to suggest that we've only seen the tip of the iceberg when it comes to the adoption of Monero—both as a standard currency of the dark web and preferred botnet monetization technique. During our research, we came across an interesting darknet market investigation wherein researchers found that while Monero is rapidly becoming the cryptocurrency of choice in the English underground economies (it is the second most commonly used, at 15 percent adoption), it still remains highly overlooked in the Eastern European markets (at only three percent). See Figure 7 for a detailed illustration of this adoption.

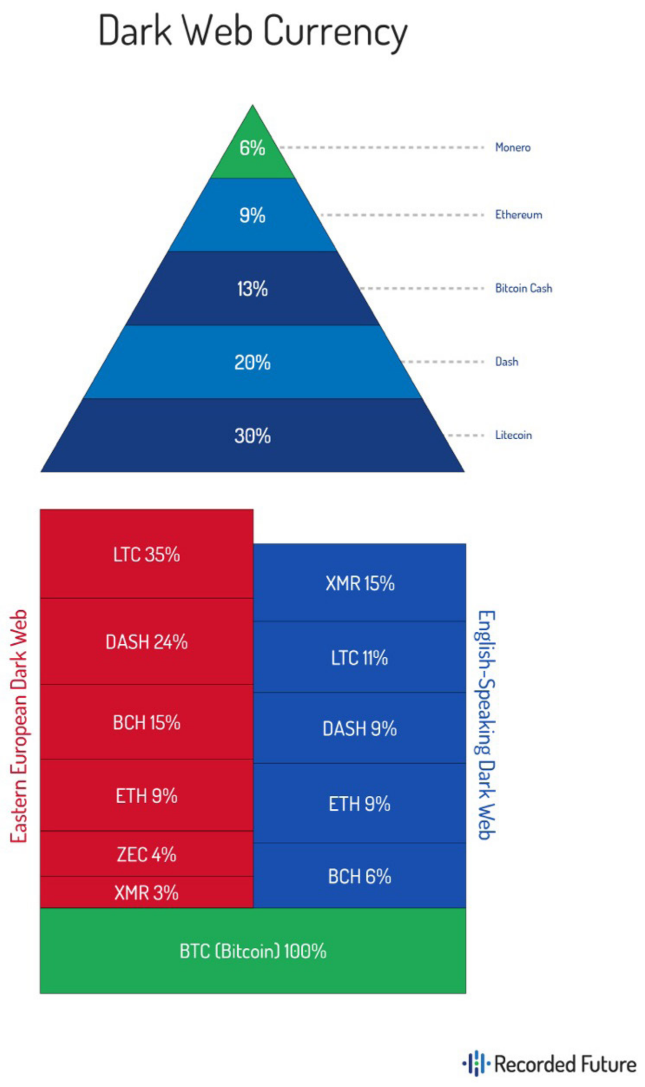


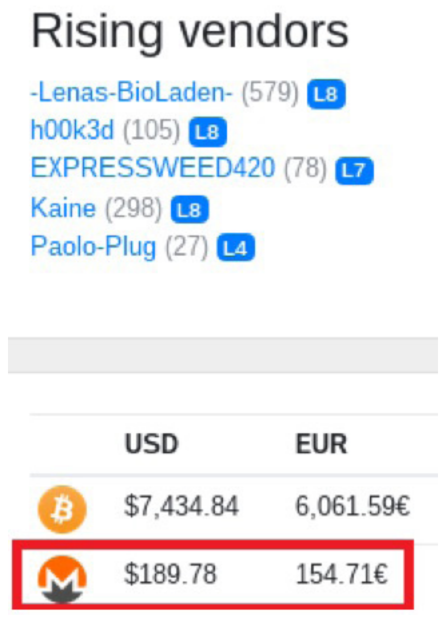
*Figure 7*

This information paints an interesting picture, due to the fact that it is the Eastern European dark web, which is known to historically operate the largest and most advanced commercial botnets, run as professional businesses by organized crime syndicates. The low adoption of Monero in Eastern Europe is surprising, but speaks more to a lack of interest in privacy or lack of awareness of XMR rather than a conscious decision to neglect it for several reasons:

1. The number one cryptocurrency in the Eastern European darknet markets is Litecoin. Litecoin is not a privacy coin and was likely adopted out of simple convenience as an alternative to Bitcoin (as it boasts significantly faster transaction times and fees) rather than a consciously chosen cryptocurrency with an emphasis on privacy to shield the buyer and seller from law enforcement.

2. Monero remains the only "true" privacy coin (through the use of stealth addresses and ring signatures) in that anonymous transactions are enabled by default (as opposed to DASH and ZCash, which are public by default), which substantially contributes to anonymity through almost total opaqueness of its blockchain.

During our research, we explored the illicit darknet website Wall St Market, known for selling everything from malware to stolen identities to illegal drugs. Sure enough, the picture painted by the aforementioned research was spot on. Right on the front page, along with their top vendors, Monero and Bitcoin are listed as the accepted currencies (See Figure 8). This is what we would expect on an English speaking darknet market.



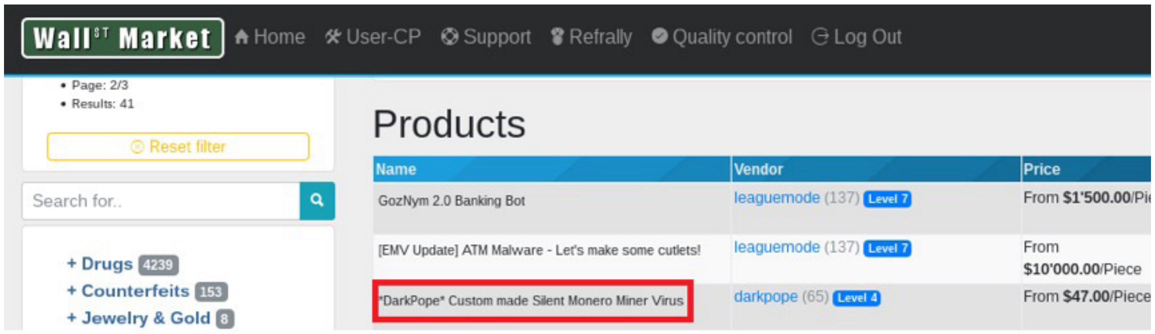
Rising vendors

-Lenas-BioLaden- (579) L8
h00k3d (105) L8
EXPRESSWEED420 (78) L7
Kaine (298) L8
Paolo-Plug (27) L4

| | USD | EUR |
|---|---|---|
| | $7,434.84 | 6,061.59€ |
| | $189.78 | 154.71€ |

*Figure 8*

Browsing through the malware products for sale, we quickly noticed a trend toward cryptominers. One of the very first popular products is a stealth Monero miner for use in botnets (See Figure 9).



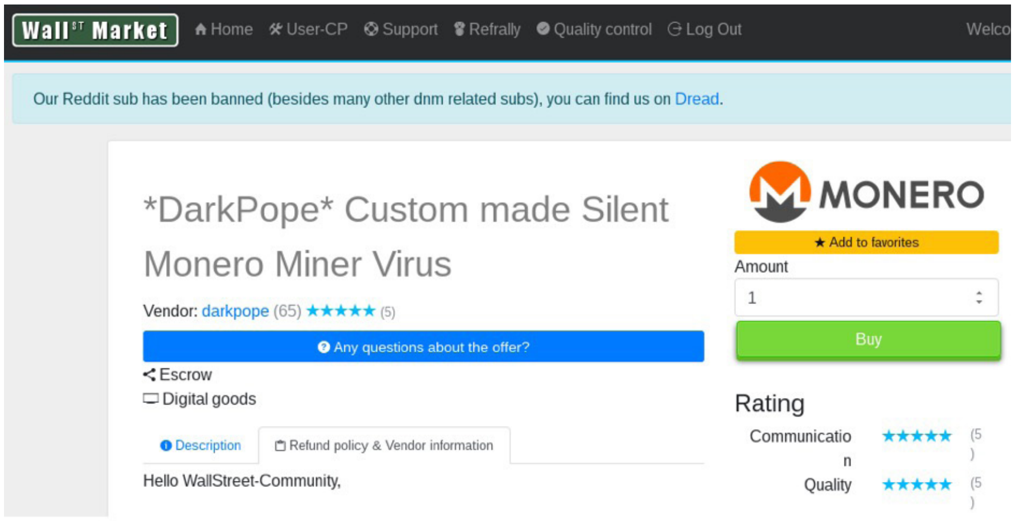
- Page: 2/3
- Results: 41

Products

| Name | Vendor | Price |
|---|---|---|
| GozNym 2.0 Banking Bot | leaguemode (137) Level 7 | From $1'500.00/Pi |
| [EMV Update] ATM Malware - Let's make some cutlets! | leaguemode (137) Level 7 | From $10'000.00/Piece |
| *DarkPope* Custom made Silent Monero Miner Virus | darkpope (65) Level 4 | From $47.00/Piece |

Figure 9

Another interesting thing to note in the screenshot above is that the Monero miner malware is listed right alongside a banking Trojan and ATM malware, which are two of the classic historical malware monetization techniques.



Our Reddit sub has been banned (besides many other dnm related subs), you can find us on Dread.

*DarkPope* Custom made Silent Monero Miner Virus

Vendor: darkpope (65) ★★★★★ (5)

Escrow
Digital goods

Hello WallStreet-Community,

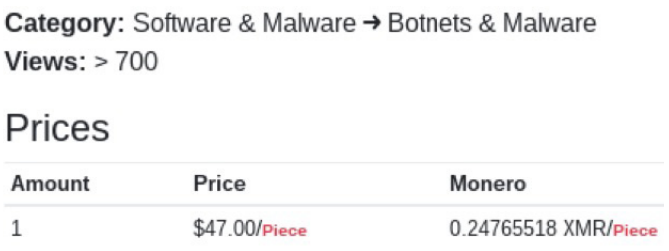
Amount: 1

Rating

| | | |
|---|---|---|
| Communication | ★★★★★ | (5) |
| Quality | ★★★★★ | (5) |

Figure 10

The vendor (fittingly) only accepts payments for his Monero mining malware in Monero (See Figure 11).



**Category:** Software & Malware ➜ Botnets & Malware
**Views:** > 700

Prices

| Amount | Price | Monero |
|---|---|---|
| 1 | $47.00/Piece | 0.24765518 XMR/Piece |

Figure 11

As Monero becomes well known in the dark web markets as the ideal privacy coin, its adoption may rapidly increase, and its prized ability to monetize botnet operations may play a major contributing role in this.

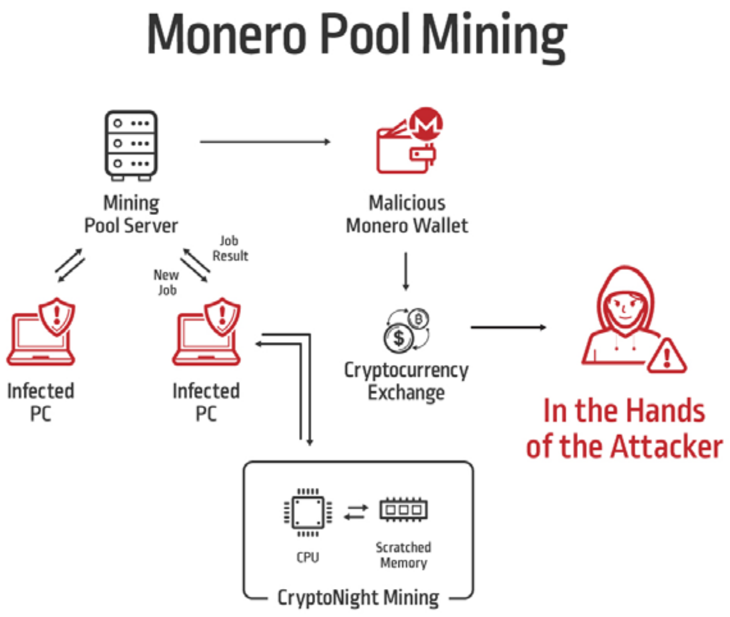
## Anatomy of the XMRig Miner

### Overview

There is a wide variety of open source cryptocurrency miner applications emerging from the open source community. A botnet operator has a very different set of concerns for his/her mining operation than a legitimate mining operation does. These concerns are:

1. Maximizing mining revenue with a very low average quality of hardware.
2. Focusing on CPU mining since this will provide a much more consistent output, as opposed to relying on graphics cards attached to compromised systems.
3. Evading detection by leaving a minimal footprint, both to security software and to the user (who may suspect an issue if their system performance is considerably degraded).
4. Simplicity and as much native implementation as possible in the design of the miner itself. This is a key concern because it is not uncommon for mining applications to have a very large (by malware standards) number of components. This bloats the miner and makes it easier to detect and cumbersome to compile from scratch due to the sheer number of third-party dependencies.

These factors coalesce into a unique problem, to which the  solution at the time of this writing appears to be optimized Monero CPU mining. The best example of this is XMRig.



XMRig is an open source Monero CPU miner published under a GNU General Public License (GPL) with support for Windows, which in and of itself is not malicious. It is largely written in C++ with some native Windows-specific components, and uses only two external library dependencies (libuv and libmicrohttpd). This miner has become increasingly popular among malware writers for the reasons outlined above. It is easy to compile, small, can function with a single executable and has Windows-specific performance optimizations.

## Source Code Structure

### Overview/Scope

The XMRig source code can be found on GitHub. Visual Studio solution files are generated using CMake, and this solution comes with four different build configurations:

1. Debug
2. Release
3. MinSizeRel
4. RelWithDebInfo

As of version 2.4.5, the release builds (Release and MinSizeRel) produce 64-bit executables of sizes 590KB and 560KB, respectively. Both come embedded with an icon, which when removed reduces the size of each by about 15KB. Compared to other miners with CryptoNight support, such as CPUMiner-multi at 1.93MB or CCMiner at 17MB, this is a very desirable size.

Note that the scope of this source code analysis does not include the following optional XMRig features:

1. The HTTP functionality (in this analysis we focus on XMRig usage through HashVault over TCP on port 5555 rather than HTTP).
2. The double hash/CryptoNightLite functionality. We focus on the plain CryptoNight logic.
3. Multiple pool functionality. We focus on the single pool logic.
4. Donation functionality. Aspects of the donation code are not discussed.

The diagram below provides a full map of the functionality of XMRig and the relationship between its major classes and components. This functionality will be explained throughout this source code analysis. Figure 12 below illustrates the anatomy of the XMRig miner from a programming and conceptual perspective.

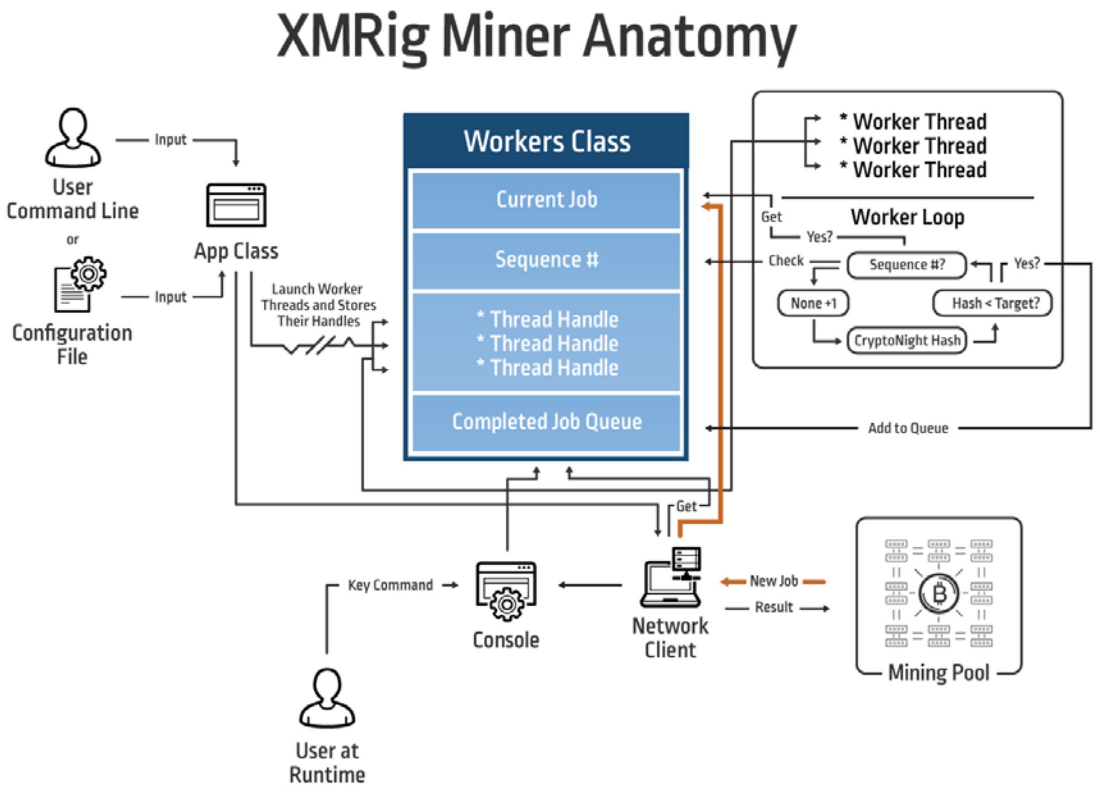
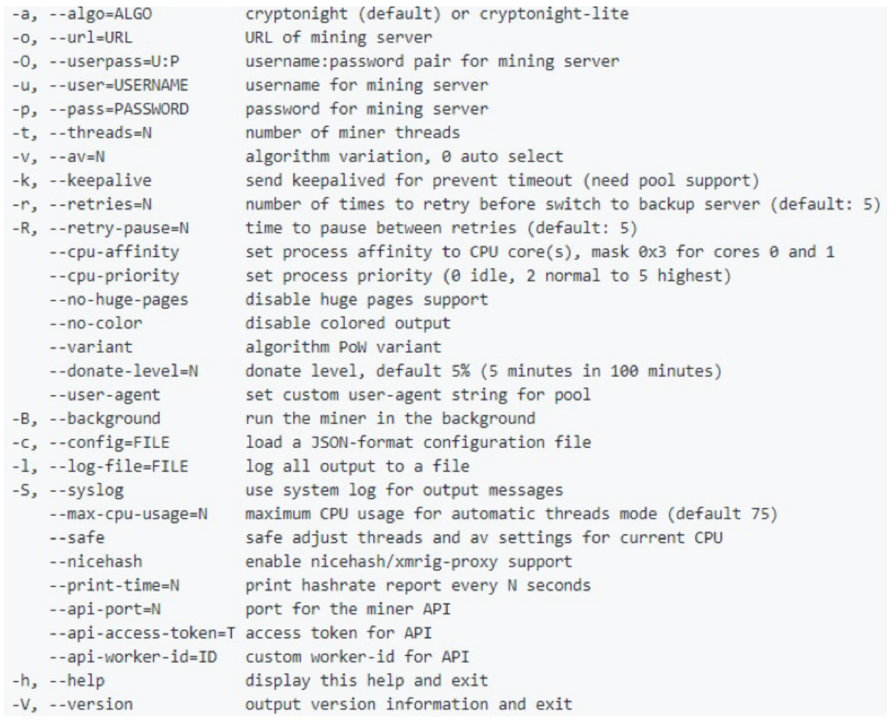


Figure 12

By examining the source code within the solution, we can piece together a general picture of the initialization procedure XMRig follows before beginning its primary mining loop.

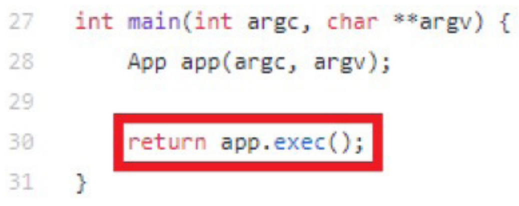
The command line usage of XMRig is documented on GitHub (shown in Figure 13 below).



```
-a, --algo=ALGO          cryptonight (default) or cryptonight-lite
-o, --url=URL            URL of mining server
-O, --userpass=U:P       username:password pair for mining server
-u, --user=USERNAME      username for mining server
-p, --pass=PASSWORD      password for mining server
-t, --threads=N          number of miner threads
-v, --av=N               algorithm variation, 0 auto select
-k, --keepalive          send keepalived for prevent timeout (need pool support)
-r, --retries=N          number of times to retry before switch to backup server (default: 5)
-R, --retry-pause=N      time to pause between retries (default: 5)
    --cpu-affinity       set process affinity to CPU core(s), mask 0x3 for cores 0 and 1
    --cpu-priority       set process priority (0 idle, 2 normal to 5 highest)
    --no-huge-pages      disable huge pages support
    --no-color           disable colored output
    --variant            algorithm PoW variant
    --donate-level=N     donate level, default 5% (5 minutes in 100 minutes)
    --user-agent         set custom user-agent string for pool
-B, --background         run the miner in the background
-c, --config=FILE        load a JSON-format configuration file
-l, --log-file=FILE      log all output to a file
-S, --syslog             use system log for output messages
    --max-cpu-usage=N    maximum CPU usage for automatic threads mode (default 75)
    --safe               safe adjust threads and av settings for current CPU
    --nicehash           enable nicehash/xmrig-proxy support
    --print-time=N       print hashrate report every N seconds
    --api-port=N         port for the miner API
    --api-access-token=T access token for API
    --api-worker-id=ID   custom worker-id for API
-h, --help               display this help and exit
-V, --version            output version information and exit
```

*Figure 13*

In xmrig.cpp, we can see the main entry point of the program (see Figure 14), which is calling a constructor to initialize its main App class using the command line arguments it was given when executed and then calling the **App.exec** method. This tells us that the App class is where the bulk of the user setting preferences and mining loop logic are stored.

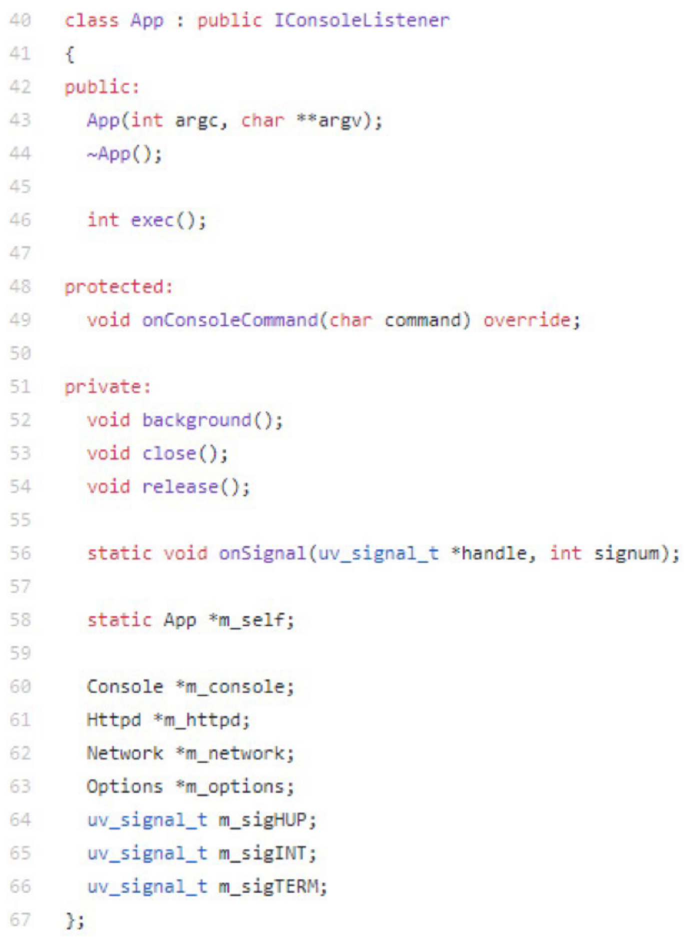


```
int main(int argc, char **argv) {
    App app(argc, argv);

    return app.exec();
}
```

*Figure 14*

## The App Class

The App class is the main entry point logic of XMRig. It is responsible for parsing user configuration settings, initializing the classes responsible for handling the behavior of the console, handling user commands, managing the interactions between the miner client and the pool, and managing the mining threads themselves. The App class also contains the primary mining loop logic that forms the base of XMRig functionality. The App.h header provides a prototype for the App class (see Figure 15).



```
class App : public IConsoleListener
{
public:
  App(int argc, char **argv);
  ~App();

  int exec();

protected:
  void onConsoleCommand(char command) override;

private:
  void background();
  void close();
  void release();

  static void onSignal(uv_signal_t *handle, int signum);

  static App *m_self;

  Console *m_console;
  Httpd *m_httpd;
  Network *m_network;
  Options *m_options;
  uv_signal_t m_sigHUP;
  uv_signal_t m_sigINT;
  uv_signal_t m_sigTERM;
};
```

*Figure 15*

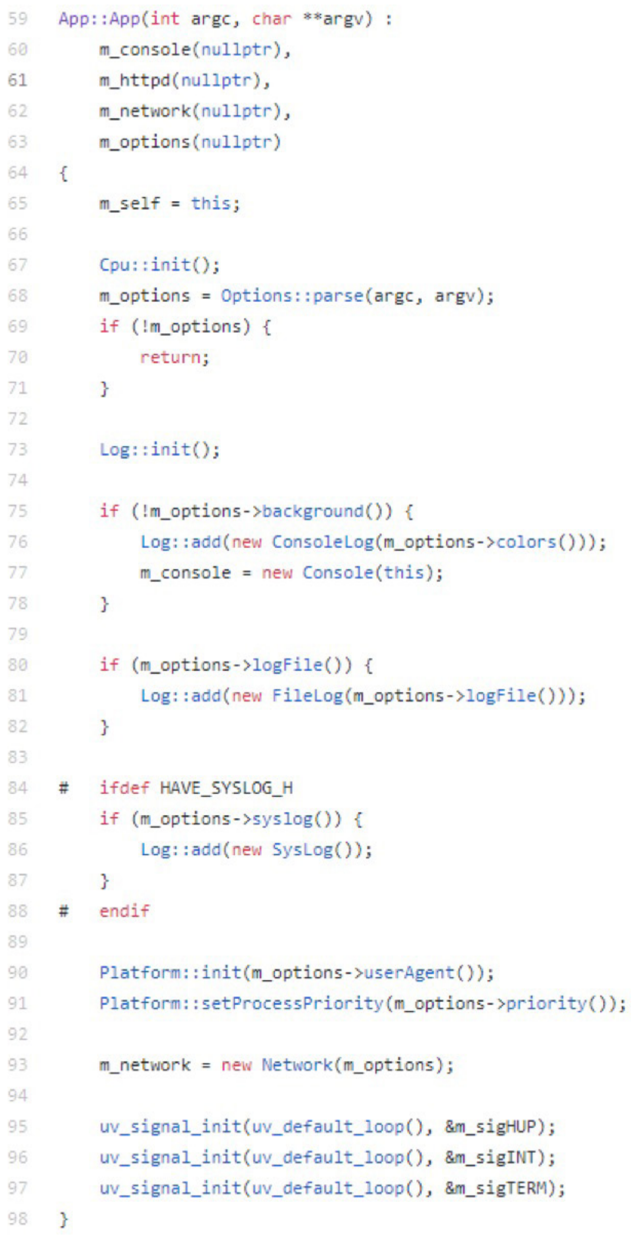
The class is pretty straightforward. It contains its own instances of several classes:

- Options (which stores configuration settings).
- Console (which handles console I/O including runtime user key commands through the third-party libuv dependency).
- Httpd (which manages the connection between the miner and the pool).
- Network (which contains the client code to send/receive JSON data from the pool and parse/handle it).
- The libuv signals (which manages ticks/timing for the mining loop logic).

It also contains a few simple methods to handle high-level application logic:

- The Constructor class to initialize the aforementioned classes based on the user-supplied configuration settings (which are stored in the Options class).
- **Exec()**, which contains the primary mining loop logic.
- **onSignal()** to interface with the libuv signals.
- **onConsoleCommand()** to handle runtime console commands from the user (such as 'p' to pause, 'h' for hashrate, 'r' for resume).
- **background()** to hide the console window (via the ShowWindow API) and run silently in the background based on the user-supplied "--background" option.

In App.c, the constructor for the App class is simplistic and easy to understand.



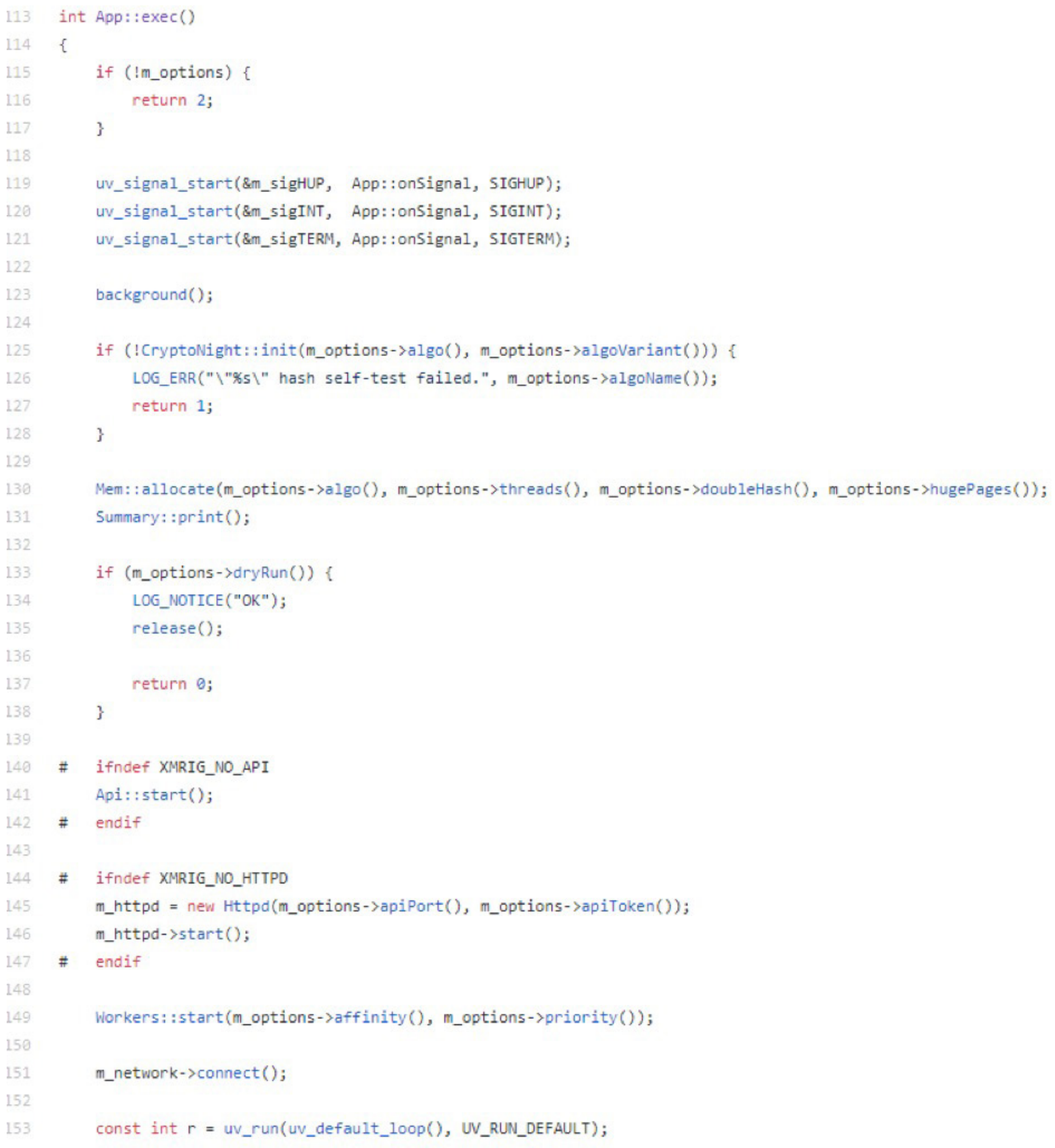
```
59  App::App(int argc, char **argv) :
60      m_console(nullptr),
61      m_httpd(nullptr),
62      m_network(nullptr),
63      m_options(nullptr)
64  {
65      m_self = this;
66
67      Cpu::init();
68      m_options = Options::parse(argc, argv);
69      if (!m_options) {
70          return;
71      }
72
73      Log::init();
74
75      if (!m_options->background()) {
76          Log::add(new ConsoleLog(m_options->colors()));
77          m_console = new Console(this);
78      }
79
80      if (m_options->logFile()) {
81          Log::add(new FileLog(m_options->logFile()));
82      }
83
84  #   ifdef HAVE_SYSLOG_H
85      if (m_options->syslog()) {
86          Log::add(new SysLog());
87      }
88  #   endif
89
90      Platform::init(m_options->userAgent());
91      Platform::setProcessPriority(m_options->priority());
92
93      m_network = new Network(m_options);
94
95      uv_signal_init(uv_default_loop(), &m_sigHUP);
96      uv_signal_init(uv_default_loop(), &m_sigINT);
97      uv_signal_init(uv_default_loop(), &m_sigTERM);
98  }
```

This constructor is initializing the App class into being the base of the user setting configuration and mining loop logic. Here is an overview of what this routine is doing:

1. Line 67 – Initializing the CPU class. This is a third-party dependency called cpuid, which provides a low-level and OS-independent CPU information enumeration interface. In the context of XMRig, its useful functionality is:
    a. To enumerate manufacturer details.
    b. To identify the cryptographic algorithms the CPU supports.
    c. To identify the optimal number of worker threads to utilize while mining.
    d. To identify the unique properties of the CPU (cache, special features) in order to give a unique optimization to the CPU mining routine on the local machine.
2. Line 68 – Parsing the user-supplied command-line arguments and configuring the Options class accordingly.

3. Line 73 – Initializing the logging capabilities (if applicable) of the program based on user input.
4. Line 90 – Initializing the miner user agent, which will be a string consisting of the Windows OS, XMRig, libuv and compiler versions.
5. Line 91 – Setting the appropriate process priority using the SetPriorityClass API. This determines how aggressive/noisy the miner will be in relation to legitimate applications run by the user. For example, by using the IDLE_PRIORITY_CLASS flag with SetPriorityClass, the miner process can be designated to only execute in idle CPU cycles (when no other application needs them).
6. Line 93– Initializing the Network class based on the user-supplied settings stored in the Options class. This class contains the client code to interact with the mining pool.
7. Line 95 - 97 – Initializing the libuv signals which will be used to handle generic exit events from the console.

Now that we understand the contents/initialization of the App class, we can look at the **App.exec()** class, which contains the main miner loop logic for XMRig (see Figure 16 below).

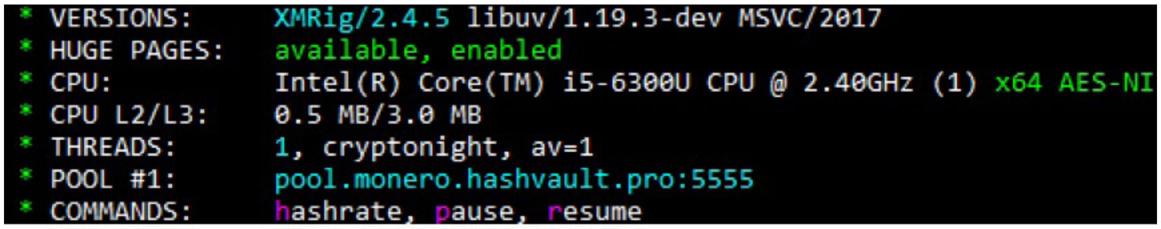


```
int App::exec()
{
    if (!m_options) {
        return 2;
    }

    uv_signal_start(&m_sigHUP,  App::onSignal, SIGHUP);
    uv_signal_start(&m_sigINT,  App::onSignal, SIGINT);
    uv_signal_start(&m_sigTERM, App::onSignal, SIGTERM);

    background();

    if (!CryptoNight::init(m_options->algo(), m_options->algoVariant())) {
        LOG_ERR("\"%s\" hash self-test failed.", m_options->algoName());
        return 1;
    }

    Mem::allocate(m_options->algo(), m_options->threads(), m_options->doubleHash(), m_options->hugePages());
    Summary::print();

    if (m_options->dryRun()) {
        LOG_NOTICE("OK");
        release();

        return 0;
    }

#   ifndef XMRIG_NO_API
    Api::start();
#   endif

#   ifndef XMRIG_NO_HTTPD
    m_httpd = new Httpd(m_options->apiPort(), m_options->apiToken());
    m_httpd->start();
#   endif

    Workers::start(m_options->affinity(), m_options->priority());

    m_network->connect();

    const int r = uv_run(uv_default_loop(), UV_RUN_DEFAULT);
```

*Figure 16*

There are a couple of interesting things happening in this function worth examining a bit closer. First, a summary of what this function is doing:

1. Lines 119-121 – Starting the libuv signals and connecting them with a callback handler (**App::onSignal**). As demonstrated by the libuv documentation for signal events, the three events being handled (SIGHUP, SIGINIT, SIGTERM) are simply signaling the termination of the console window via Ctrl+C, closing the window, or Ctrl+break. Therefore, App::onSignal is simply a cleanup handler.
2. Line 123 – Hiding the primary console window via the ShowWindow API based on the "--background" argument.
3. Line 125 – Initializing the CryptoNight algorithm. This is the primary PoW algorithm utilized by Monero and the most essential component of this miner application.
4. Line 130 – Allocating the virtual memory, which will be used as the scratchpad space for the miner threads while utilizing CryptoNight.
5. Line 131 – Printing the primary XMRig usage info to the console window. This corresponds to the output we see when we initially start up the miner (see Figure 18 below).



```
 * VERSIONS:     XMRig/2.4.5 libuv/1.19.3-dev MSVC/2017
 * HUGE PAGES:   available, enabled
 * CPU:          Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz (1) x64 AES-NI
 * CPU L2/L3:    0.5 MB/3.0 MB
 * THREADS:      1, cryptonight, av=1
 * POOL #1:      pool.monero.hashvault.pro:5555
 * COMMANDS:     hashrate, pause, resume
```

*Figure 18*

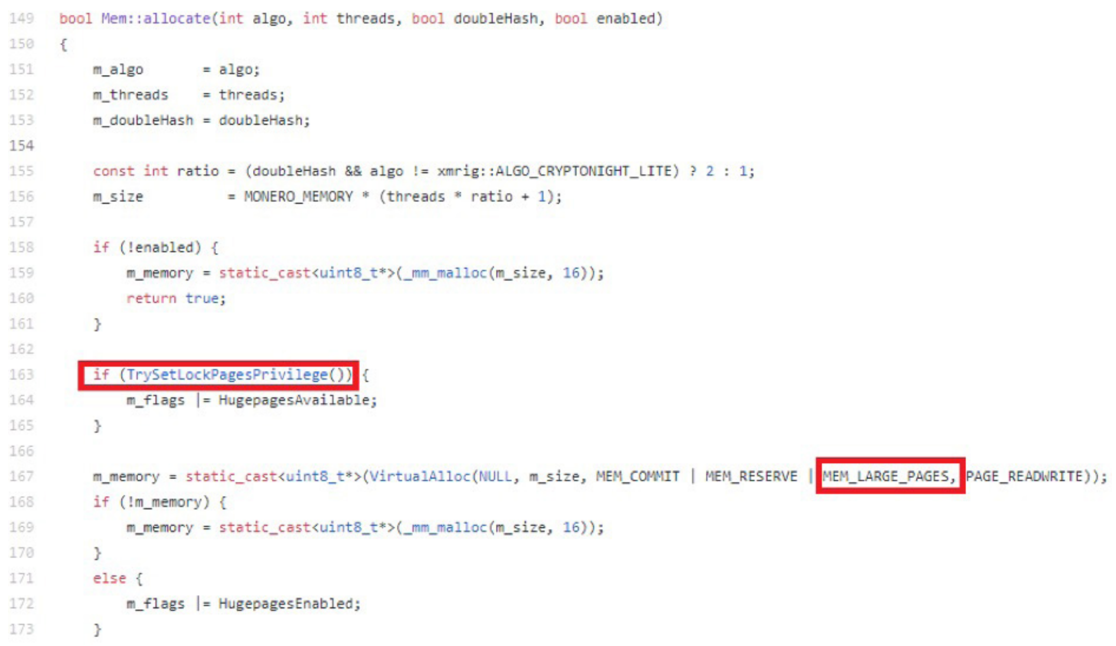
6. Line 149 – Starting the worker threads responsible for doing the actual mining and passing their results to the client.
7. Line 151 – Connecting the client (within the Network class) to the pool. Note that is not necessarily related to the Httpd functionality started on lines 145-146 (and the potential relationship won't be analyzed here).
8. Line 153 – Starting the libuv tick loops which were linked to the **App::onSignal** method on lines 119-121.

## Large-Page Virtual Memory

A unique facet of Monero mining is the intentional emphasis on memory latency as a mechanism to thwart ASIC miners. Large-page support is a feature of the Windows VirtualAlloc function, which according to MSDN, works as follows: "Each large-page translation uses a single translation buffer inside the CPU. The size of this buffer is typically three orders of magnitude larger than the native page size; this increases the efficiency of the translation buffer, which can increase performance for frequently accessed memory." The result of this is that by locking its virtual ("scratchpad") memory in place, a Monero miner can significantly optimize the speed of its CryptoNight implementation (on average as much as 20 percent) due to the high emphasis on memory latency speeds in this algorithm.

An obstacle to the implementation of this is that the privilege needed to lock virtual memory in this way requires a special user account privilege that is not enabled by default (and requires administrative privileges, UAC elevation and a reboot to achieve). A manual description of its modification can be found here.

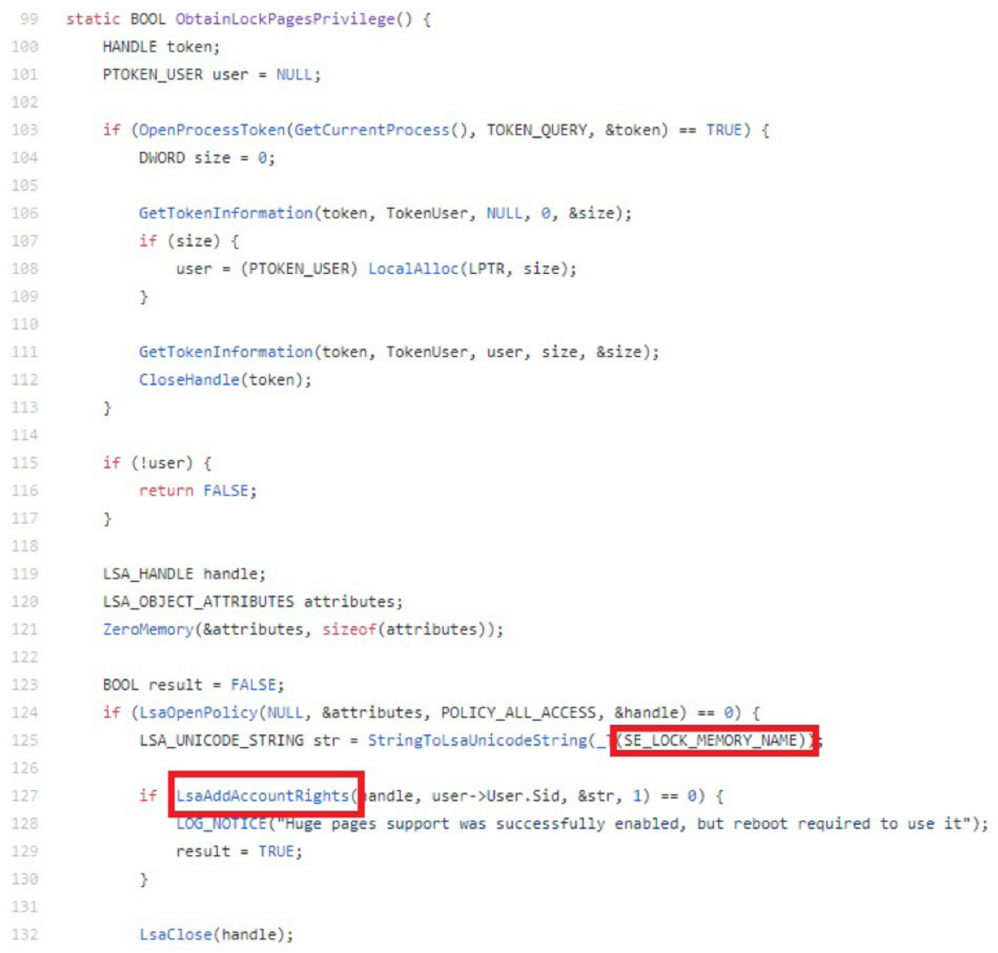
XMRig solves this problem in an automated way by utilizing the native Windows API. Within Mem_win.cpp we can see the attempted allocation of the virtual memory for CryptoNight using this special flag (the MEM_LARGE_PAGES flag). See the highlighted portions of Figure 19 below.



```
bool Mem::allocate(int algo, int threads, bool doubleHash, bool enabled)
{
    m_algo       = algo;
    m_threads    = threads;
    m_doubleHash = doubleHash;

    const int ratio = (doubleHash && algo != xmrig::ALGO_CRYPTONIGHT_LITE) ? 2 : 1;
    m_size          = MONERO_MEMORY * (threads * ratio + 1);

    if (!enabled) {
        m_memory = static_cast<uint8_t*>(_mm_malloc(m_size, 16));
        return true;
    }

    if (TrySetLockPagesPrivilege()) {
        m_flags |= HugepagesAvailable;
    }

    m_memory = static_cast<uint8_t*>(VirtualAlloc(NULL, m_size, MEM_COMMIT | MEM_RESERVE | MEM_LARGE_PAGES, PAGE_READWRITE));
    if (!m_memory) {
        m_memory = static_cast<uint8_t*>(_mm_malloc(m_size, 16));
    }
    else {
        m_flags |= HugepagesEnabled;
    }
```

*Figure 19*

The key details highlighted in the code in Figure 19 are the use of the MEM_LARGE_PAGES flag passed to VirtualAlloc on line 167, and the call to TrySetLockPrivilege on line 163, the underlying source code for which is as follows in Figure 20:



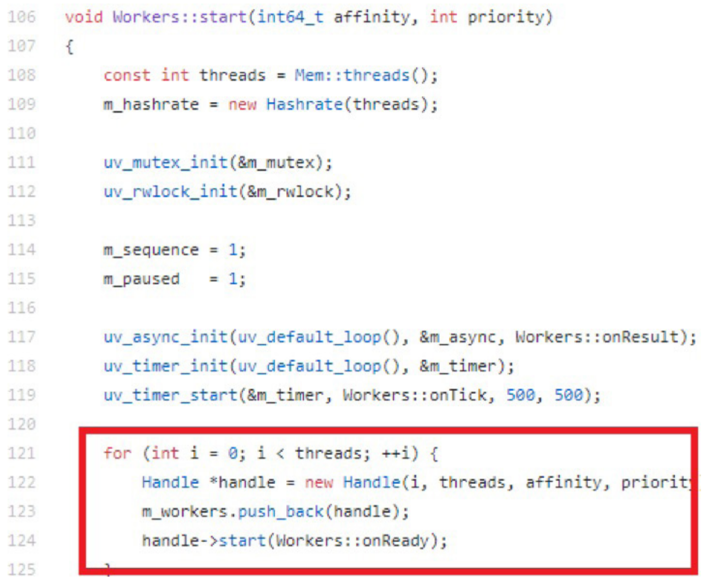
```
static BOOL ObtainLockPagesPrivilege() {
    HANDLE token;
    PTOKEN_USER user = NULL;

    if (OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &token) == TRUE) {
        DWORD size = 0;

        GetTokenInformation(token, TokenUser, NULL, 0, &size);
        if (size) {
            user = (PTOKEN_USER) LocalAlloc(LPTR, size);
        }

        GetTokenInformation(token, TokenUser, user, size, &size);
        CloseHandle(token);
    }

    if (!user) {
        return FALSE;
    }

    LSA_HANDLE handle;
    LSA_OBJECT_ATTRIBUTES attributes;
    ZeroMemory(&attributes, sizeof(attributes));

    BOOL result = FALSE;
    if (LsaOpenPolicy(NULL, &attributes, POLICY_ALL_ACCESS, &handle) == 0) {
        LSA_UNICODE_STRING str = StringToLsaUnicodeString(_T(SE_LOCK_MEMORY_NAME));

        if (LsaAddAccountRights(handle, user->User.Sid, &str, 1) == 0) {
            LOG_NOTICE("Huge pages support was successfully enabled, but reboot required to use it");
            result = TRUE;
        }

        LsaClose(handle);
```

What this code is doing is opening a handle to the current miner process token, and then using this token to extract the username info (in the form of a TOKEN_USER struct, which contains the user SID, among other details) for which to enhance privileges. With the user info, the function then utilizes the LsaOpenPolicy (which will require UAC elevation) and LsaAddAccountRights Windows API functions to add the SE_LOCK_MEMORY_NAME privilege to this user. The current user will then be able to successfully call VirtualAlloc with the MEM_LARGE_PAGES flag after a reboot.

## Workers

The core functionality of XMRig lies within its worker threads, which are responsible for accepting input in the form of data from the mining pool (a data blob and target value), and output in the form of CryptoNight hashes and nonces (to be sent back to the mining pool). The rest of the XMRig code is simply encompassing these worker threads and allowing them to interact with the user and mining pool.

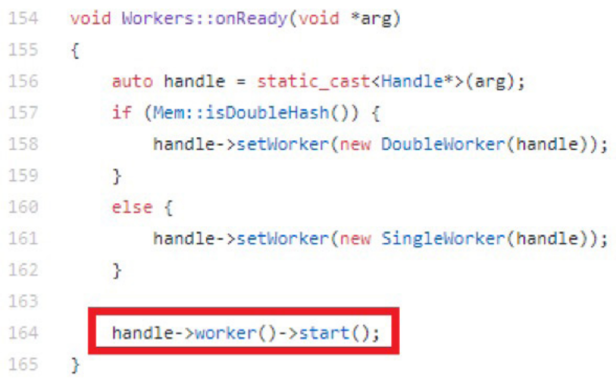
The worker code in XMRig is divided into the Workers class prototyped in Workers.h and the SingleWorker class in SingleWorker.cpp (which inherits from the generic Worker class). The Workers class contains high-level information describing the overall status of all of the worker threads in the application (each of which has its own SingleWorker class). The worker threads are initialized through the Workers class in the **Workers::start** method (see Figure 20 on next page).



```
void Workers::start(int64_t affinity, int priority)
{
    const int threads = Mem::threads();
    m_hashrate = new Hashrate(threads);

    uv_mutex_init(&m_mutex);
    uv_rwlock_init(&m_rwlock);

    m_sequence = 1;
    m_paused   = 1;

    uv_async_init(uv_default_loop(), &m_async, Workers::onResult);
    uv_timer_init(uv_default_loop(), &m_timer);
    uv_timer_start(&m_timer, Workers::onTick, 500, 500);

    for (int i = 0; i < threads; ++i) {
        Handle *handle = new Handle(i, threads, affinity, priorit
        m_workers.push_back(handle);
        handle->start(Workers::onReady);
    }
```

Figure 20

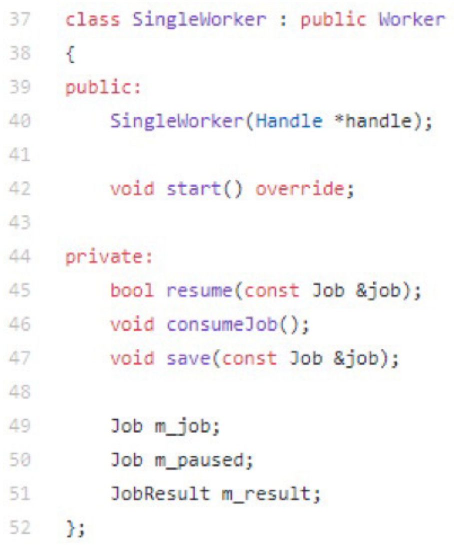
Threads (the number of which is supplied through user input or lack thereof) are created and the handles of each are stored in the Workers class. Each thread is initially launched through the **Workers::onReady** method (as illustrated by the highlighted portion of Figure 21 below).



```
154  void Workers::onReady(void *arg)
155  {
156      auto handle = static_cast<Handle*>(arg);
157      if (Mem::isDoubleHash()) {
158          handle->setWorker(new DoubleWorker(handle));
159      }
160      else {
161          handle->setWorker(new SingleWorker(handle));
162      }
163
164      handle->worker()->start();
165  }
```

Figure 21

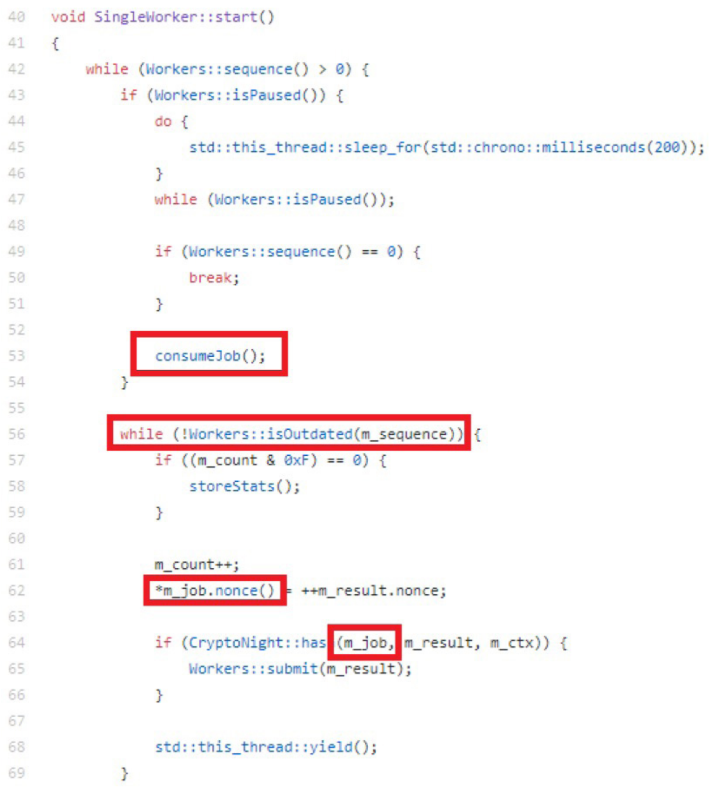
The most interesting part of this function is the setWorker method of the handle class called out on line 161. This is where each thread is assigned its own SingleWorker class to store its own individual mining state. See Figure 22 for a prototype of the SingleWorker class.



```
class SingleWorker : public Worker
{
public:
    SingleWorker(Handle *handle);

    void start() override;

private:
    bool resume(const Job &job);
    void consumeJob();
    void save(const Job &job);

    Job m_job;
    Job m_paused;
    JobResult m_result;
};
```

Figure 22

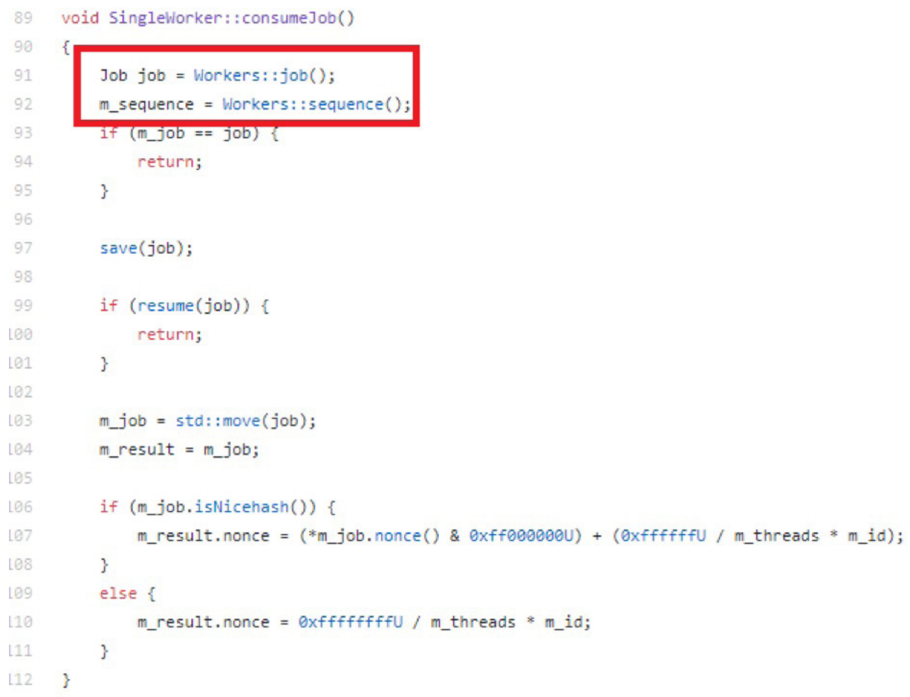
The start method for each SingleWorker class thread contains the central mining logic loop of the entire XMRig application. The **SingleWorker::start** method consists of two loops, one within the other (see Figure 23 below. The key aspects of the mining loop logic have been highlighted).



```
void SingleWorker::start()
{
    while (Workers::sequence() > 0) {
        if (Workers::isPaused()) {
            do {
                std::this_thread::sleep_for(std::chrono::milliseconds(200));
            }
            while (Workers::isPaused());

            if (Workers::sequence() == 0) {
                break;
            }

            consumeJob();
        }

        while (!Workers::isOutdated(m_sequence)) {
            if ((m_count & 0xF) == 0) {
                storeStats();
            }

            m_count++;
            *m_job.nonce() = ++m_result.nonce;

            if (CryptoNight::hash(m_job, m_result, m_ctx)) {
                Workers::submit(m_result);
            }

            std::this_thread::yield();
        }
```

Figure 23

The first layer of the loop is essentially just grabbing new job data from the miner pool via the **consumeJob** function. This function is a method within the local SingleWorker class (see Figure 24 below).

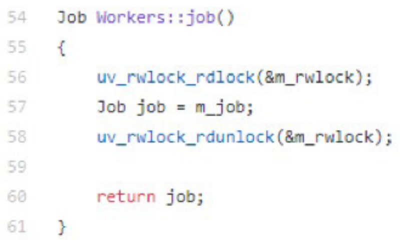


```
89  void SingleWorker::consumeJob()
90  {
91      Job job = Workers::job();
92      m_sequence = Workers::sequence();
93      if (m_job == job) {
94          return;
95      }
96
97      save(job);
98
99      if (resume(job)) {
100         return;
101     }
102
103     m_job = std::move(job);
104     m_result = m_job;
105
106     if (m_job.isNicehash()) {
107         m_result.nonce = (*m_job.nonce() & 0xff000000U) + (0xffffffU / m_threads * m_id);
108     }
109     else {
110         m_result.nonce = 0xffffffffU / m_threads * m_id;
111     }
112 }
```

*Figure 24*

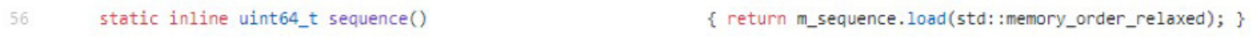
This function is key to understanding the way in which XMRig is able to take JSON data from the miner pool and distribute it to workers for processing. The essential logic behind this function is as follows:

1. Line 91 – Obtains a copy of the most recent Job object from the Workers class. The Job object (an instance of the Job class in Job.cpp) will ultimately contain the data to be hashed, the thread assigned to process it, and other details. The job method in the Workers class obtains a copy of the most recent Job object synchronized via mutex.



```
Job Workers::job()
{
    uv_rwlock_rdlock(&m_rwlock);
    Job job = m_job;
    uv_rwlock_rdunlock(&m_rwlock);

    return job;
}
```

2. Line 92 – Retrieves the sequence value from the central Workers class.



```
static inline uint64_t sequence()                      { return m_sequence.load(std::memory_order_relaxed); }
```

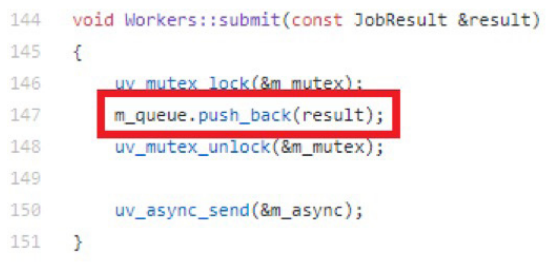
The sequence value (**Workers::m_sequence**) is an atomically synchronized 64-bit integer that stores the number of jobs processed since XMRig was started. Its primary use is within the **Workers::isOutdated** method, which is used within the secondary **SingleWorker::start** loop in order to compare the local m_sequence value within each Worker class (corresponding to each worker thread) with the central m_sequence value in the Workers class in order to determine whether or not the current worker thread is up to date with the most recent job from the mining pool. If it is not up to date, the worker thread will update its sequence count and begin attempting to solve the data blob provided within the most recent job.

3. Lines 93-95 – Checks the job object obtained from the central Workers class against the local job object stored in each Worker class to eliminate duplicate jobs.

4. Line 97 – Saves the current job object to the m_pause object to be used for restoration after pausing/unpausing.
5. Lines 103-112 – Resets the current job object for the current worker thread to the newly obtained job object and initializes a pseudo-random nonce to be used for solving the CryptoNight PoW on the job data blob. This nonce initialization is done using the worker thread ID (stored in **m_id** in the Worker class constructor) and thread count. Neither are strong random values, but they will still be unique for each thread, which is enough from the perspective of the miner since it doesn't want each worker thread to be duplicating one another's efforts when nonces happen to collide.

Once the job has been consumed into the current worker thread by the **consumeJob** method, the second layer loop within **SingleWorker::start** begins attempting to solve the PoW for the current job data blob. Each iteration of the loop, it increments the job nonce, attempting to find a valid PoW hash for the job data. The validity of a given nonce is determined within the **CryptoNight::hash** method using the target value provided by the mining pool. An in-depth explanation of the logic behind **CryptoNight::hash** can be found here.

If a valid nonce is found, **Workers::submit** is used to push the completed job object (which will now contain the nonce which solved the PoW) onto a global queue (synchronized via mutex) in the Workers class. This queue is routinely checked by the network client in order to submit PoW results to the mining pool (see Figure 25 below).



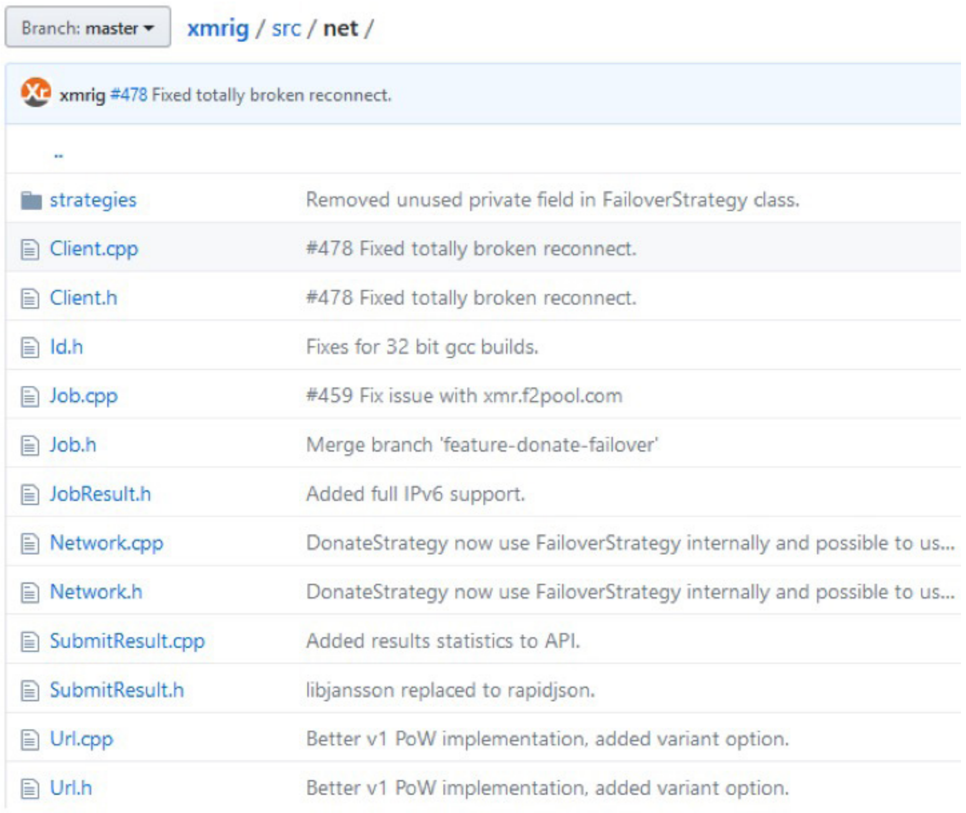
```
void Workers::submit(const JobResult &result)
{
    uv_mutex_lock(&m_mutex);
    m_queue.push_back(result);
    uv_mutex_unlock(&m_mutex);

    uv_async_send(&m_async);
}
```

*Figure 25*

## Network Client

The Network class, as its name would imply, is used to negotiate the network I/O between XMRig and the user-supplied mining pool. Its purpose could best be described as:

1. Acting as an interface between the console UI and the mining pool Client class. The Network class provides all the standard console output for XMRig, such as displaying when a new job is received from the pool, when a PoW share is accepted by the pool, and when a share is rejected by the pool.
2. Initializing the Client class, encompassed within the "Strategy" class (in this case SinglePoolStrategy), which is also initialized within the Network class.
3. Initiating the connection to the mining pool based on user-supplied input options.
4. Passing new job data to the global Workers class (to be synchronously stored in **Workers::m_job** and grabbed by worker threads as their sequence numbers become outdated).



Branch: master ▾ xmrig / src / net /

xmrig #478 Fixed totally broken reconnect.

..

| | |
|---|---|
| strategies | Removed unused private field in FailoverStrategy class. |
| Client.cpp | #478 Fixed totally broken reconnect. |
| Client.h | #478 Fixed totally broken reconnect. |
| Id.h | Fixes for 32 bit gcc builds. |
| Job.cpp | #459 Fix issue with xmr.f2pool.com |
| Job.h | Merge branch 'feature-donate-failover' |
| JobResult.h | Added full IPv6 support. |
| Network.cpp | DonateStrategy now use FailoverStrategy internally and possible to us... |
| Network.h | DonateStrategy now use FailoverStrategy internally and possible to us... |
| SubmitResult.cpp | Added results statistics to API. |
| SubmitResult.h | libjansson replaced to rapidjson. |
| Url.cpp | Better v1 PoW implementation, added variant option. |
| Url.h | Better v1 PoW implementation, added variant option. |

To initialize the Client class, the Network class begins by creating a new SinglePoolStrategy class, which constructs and stores a new Client class within its own internal m_client variable. Figure 26 below shows the constructor class for the SinglePoolStrategy class.

```
SinglePoolStrategy::SinglePoolStrategy(const Url *url, int retryPause, IStrategyListener *listener, bool quiet) :
    m_active(false),
    m_listener(listener)
{
    m_client = new Client(0, Platform::userAgent(), this);
    m_client->setUrl(url);
    m_client->setRetryPause(retryPause * 1000);
    m_client->setQuiet(quiet);
}
```

*Figure 26*

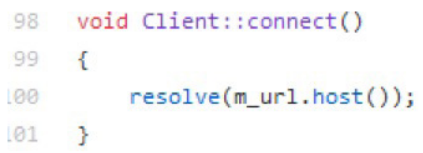
The Network class provides a path to initiate a connection to the mining pool through its connect method, which as previously described in the analysis of the App class is called through App::exec.

```
void Network::connect()
{
    m_strategy->connect();
}
```

The Strategy class has its own connect method.

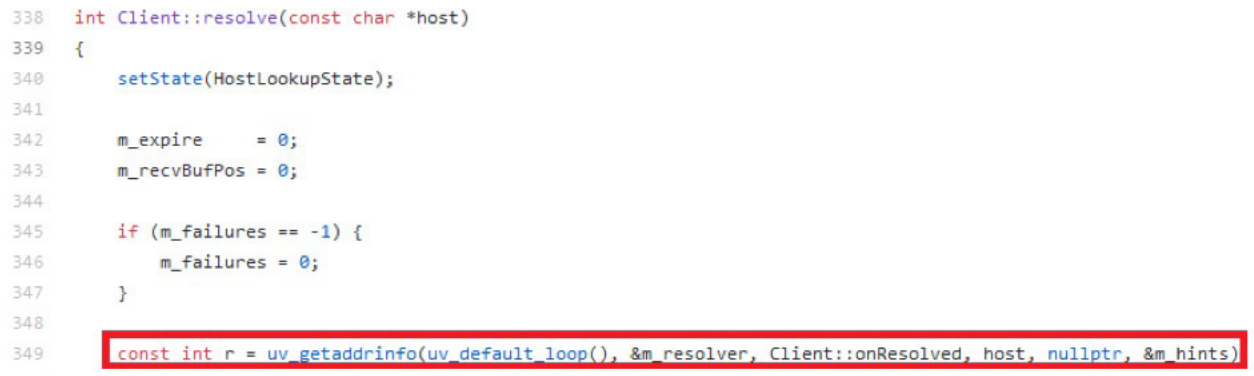
```
void SinglePoolStrategy::connect()
{
    m_client->connect();
}
```

Which, in turn, finally calls the **Client::connect** method.



```
void Client::connect()
{
    resolve(m_url.host());
}
```

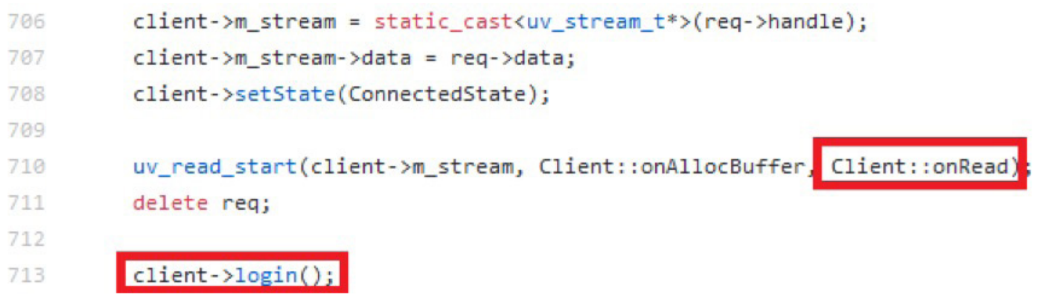
The Client::resolve method uses libuv to set **Client::onResolved** as a callback for when the mining pool address is successfully resolved from its host name (see Figure 27 below).



```
int Client::resolve(const char *host)
{
    setState(HostLookupState);

    m_expire     = 0;
    m_recvBufPos = 0;

    if (m_failures == -1) {
        m_failures = 0;
    }

    const int r = uv_getaddrinfo(uv_default_loop(), &m_resolver, Client::onResolved, host, nullptr, &m_hints);
```

*Figure 27*

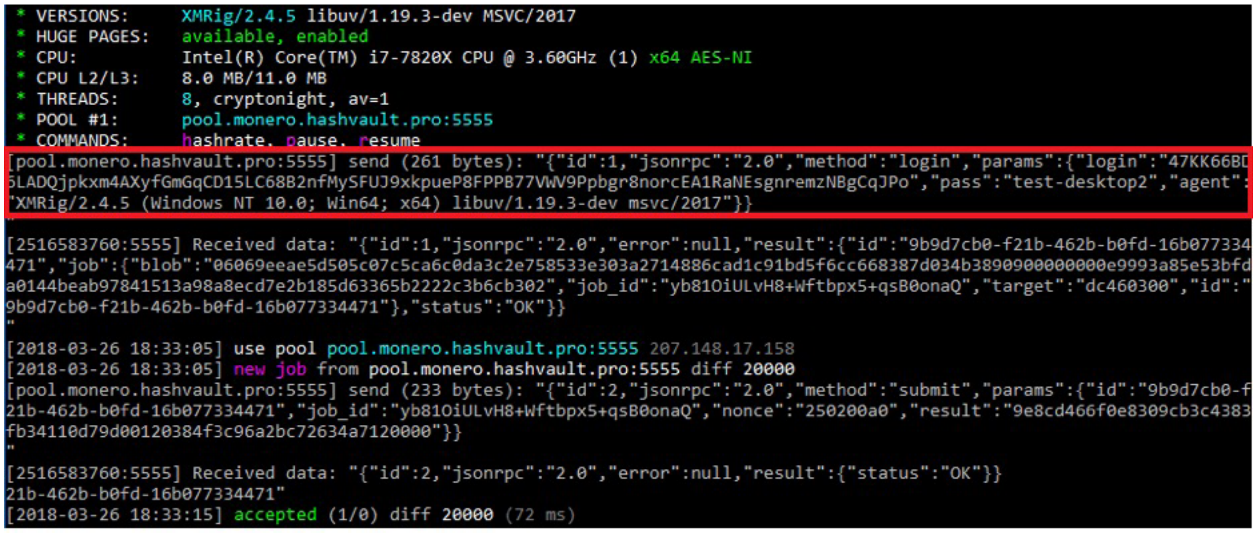
The **Client::onResolved** method uses the IP address resolved from the mining pool host name to call a second **Client::connect** overload, which in turn, makes the actual connection to the mining pool (in our case, pool.monero.hashvault.pro on port 5555) and sets **Client::onConnect** as a callback for when this connection is finished. The **Client::onConnect** method then initializes the network stream to receive incoming data, sets a callback handler to **Client::onRead** that will be called whenever new data is received from the mining pool, and finally, begins the mining pool protocol handshake by sending a login request through JSON (see Figure 28 below).



```
    client->m_stream = static_cast<uv_stream_t*>(req->handle);
    client->m_stream->data = req->data;
    client->setState(ConnectedState);

    uv_read_start(client->m_stream, Client::onAllocBuffer, Client::onRead);
    delete req;

    client->login();
```

*Figure 28*

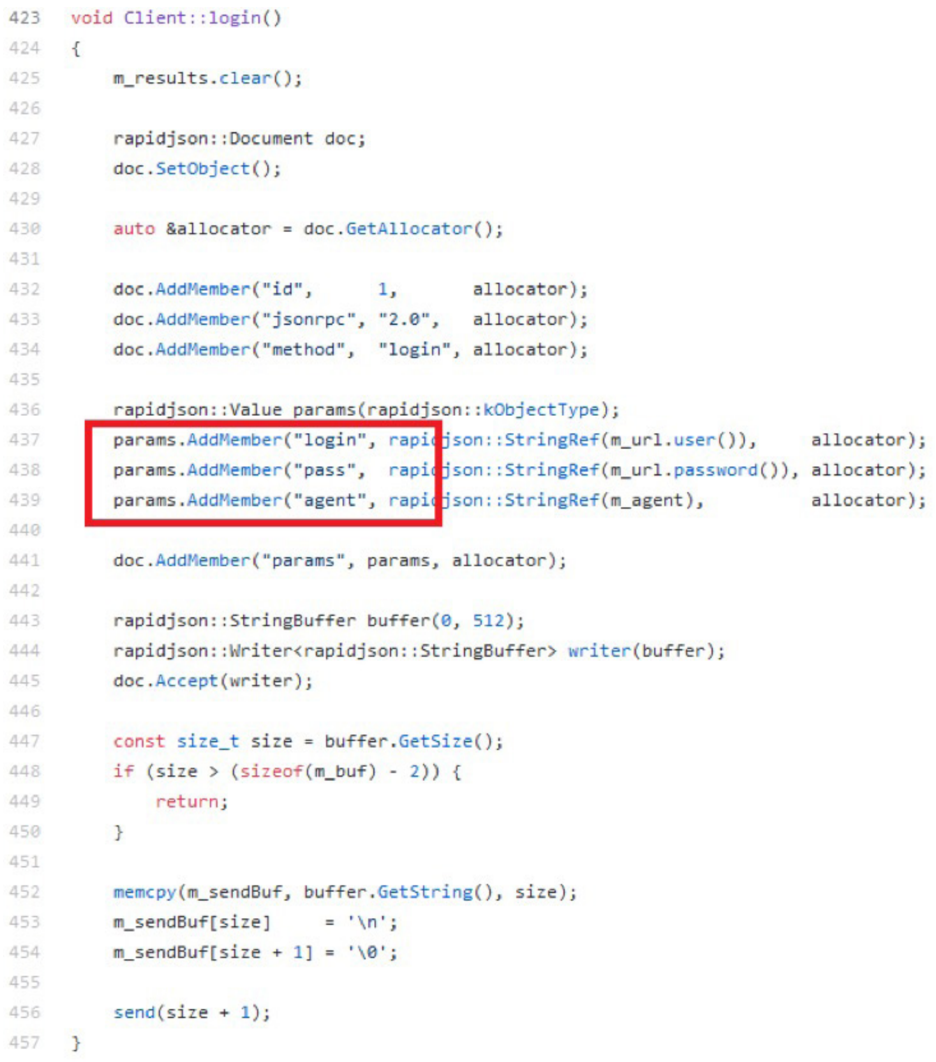
During our analysis of XMRig, we modified its source to output all incoming/outbound data to/from the mining pool to the console (through the **Client::send** and **Client::onRead** methods) to better visualize the mining pool protocol traffic (see Figure 29 below).



```
 * VERSIONS:     XMRig/2.4.5 libuv/1.19.3-dev MSVC/2017
 * HUGE PAGES:   available, enabled
 * CPU:          Intel(R) Core(TM) i7-7820X CPU @ 3.60GHz (1) x64 AES-NI
 * CPU L2/L3:    8.0 MB/11.0 MB
 * THREADS:      8, cryptonight, av=1
 * POOL #1:      pool.monero.hashvault.pro:5555
 * COMMANDS:     hashrate, pause, resume
[pool.monero.hashvault.pro:5555] send (261 bytes): "{"id":1,"jsonrpc":"2.0","method":"login","params":{"login":"47KK66BD
5LADQjpkxm4AXyfGmGqCD15LC68B2nfMySFUJ9xkpueP8FPPB77VWV9Ppbgr8norcEA1RaNEsgnremzNBgCqJPo","pass":"test-desktop2","agent":
"XMRig/2.4.5 (Windows NT 10.0; Win64; x64) libuv/1.19.3-dev msvc/2017"}}
"
[2516583760:5555] Received data: "{"id":1,"jsonrpc":"2.0","error":null,"result":{"id":"9b9d7cb0-f21b-462b-b0fd-16b077334
471","job":{"blob":"06069eeae5d505c07c5ca6c0da3c2e758533e303a2714886cad1c91bd5f6cc668387d034b3890900000000e9993a85e53bfd
a0144beab97841513a98a8ecd7e2b185d63365b2222c3b6cb302","job_id":"yb81OiULvH8+Wftbpx5+qsB0onaQ","target":"dc460300","id":"
9b9d7cb0-f21b-462b-b0fd-16b077334471"},"status":"OK"}}
"
[2018-03-26 18:33:05] use pool pool.monero.hashvault.pro:5555 207.148.17.158
[2018-03-26 18:33:05] new job from pool.monero.hashvault.pro:5555 diff 20000
[pool.monero.hashvault.pro:5555] send (233 bytes): "{"id":2,"jsonrpc":"2.0","method":"submit","params":{"id":"9b9d7cb0-f
21b-462b-b0fd-16b077334471","job_id":"yb81OiULvH8+Wftbpx5+qsB0onaQ","nonce":"250200a0","result":"9e8cd466f0e8309cb3c4383
fb34110d79d00120384f3c96a2bc72634a7120000"}}
"
[2516583760:5555] Received data: "{"id":2,"jsonrpc":"2.0","error":null,"result":{"status":"OK"}}
21b-462b-b0fd-16b077334471"
[2018-03-26 18:33:15] accepted (1/0) diff 20000 (72 ms)
```

Figure 29

By examining the source of the **Client::login** method, we can see the code responsible for generating the JSON data for the login request from XMRig.



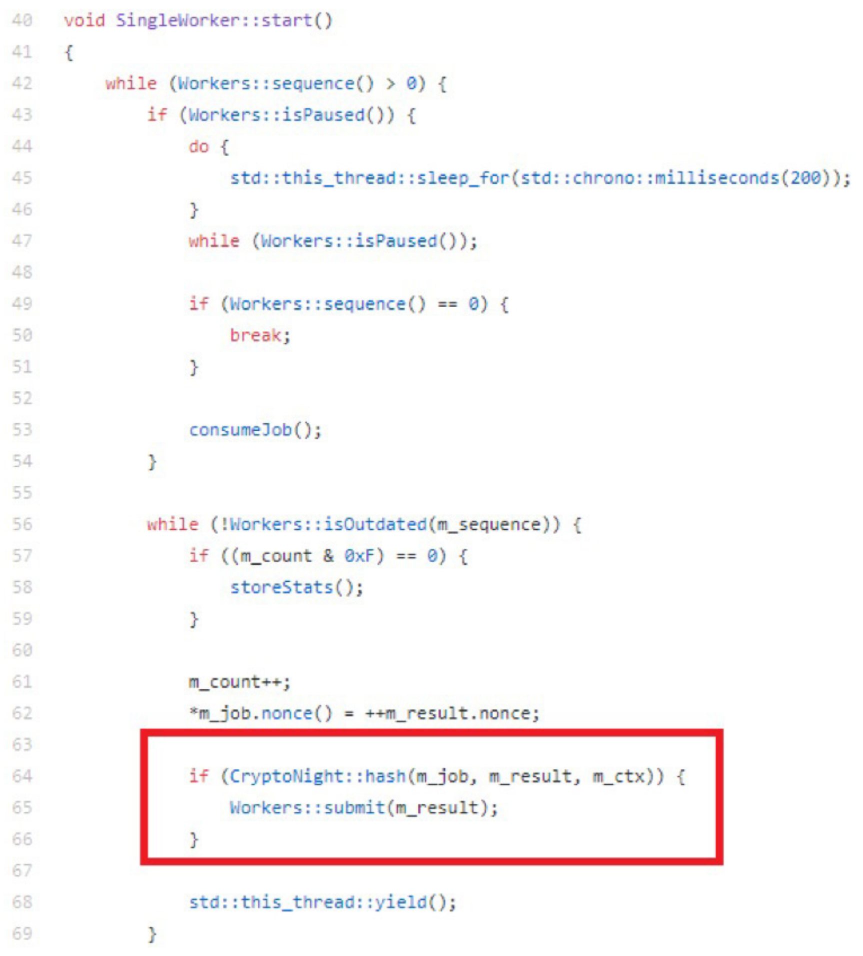
```
void Client::login()
{
    m_results.clear();

    rapidjson::Document doc;
    doc.SetObject();

    auto &allocator = doc.GetAllocator();

    doc.AddMember("id",       1,       allocator);
    doc.AddMember("jsonrpc", "2.0",    allocator);
    doc.AddMember("method",  "login",  allocator);

    rapidjson::Value params(rapidjson::kObjectType);
    params.AddMember("login", rapidjson::StringRef(m_url.user()),     allocator);
    params.AddMember("pass",  rapidjson::StringRef(m_url.password()), allocator);
    params.AddMember("agent", rapidjson::StringRef(m_agent),          allocator);

    doc.AddMember("params", params, allocator);

    rapidjson::StringBuffer buffer(0, 512);
    rapidjson::Writer<rapidjson::StringBuffer> writer(buffer);
    doc.Accept(writer);

    const size_t size = buffer.GetSize();
    if (size > (sizeof(m_buf) - 2)) {
        return;
    }

    memcpy(m_sendBuf, buffer.GetString(), size);
    m_sendBuf[size]     = '\n';
    m_sendBuf[size + 1] = '\0';

    send(size + 1);
}
```

Note how the JSON document being generated ultimately corresponds to the first sent network traffic in the screenshot above (it is the only JSON protocol traffic that contains the "login," "pass" and "agent" fields, as well as a "method" value of "login").

The last bit of interesting code in the Client class is within **Client::parseNotification** (called as a result of **Client::onRead** when raw data is received from the pool). This method will ultimately call **SinglePoolStrategy::onJobReceived** (if the JSON "method" is "job"), which connects the XMRig logic round-circle back to the previously discussed Workers class by passing a (newly generated) Job class object to **Workers::setJob** method. As previously discussed, **Workers::setJob** makes a new Job object available to the worker/miner threads via a (mutex synchronized) global variable called **m_job**, signaling worker threads to consume this Job data by incrementing its sequence count.

## CryptoNight

As discussed in a prior section, the core mining logic of XMRig is encompassed within a double layered loop in the **SingleWorker::start** method in SingleWorker.cpp. After initializing a pseudo-random nonce based on the worker thread ID and total thread count, this nonce is incremented in an endless loop (attempting to produce a valid PoW hash from the **CryptoNight::hash** method) until the current job becomes outdated (as indicated by the sequence number in the Workers class). See the highlighted portion of the SingleWorker::start class highlighted in Figure 30 below.



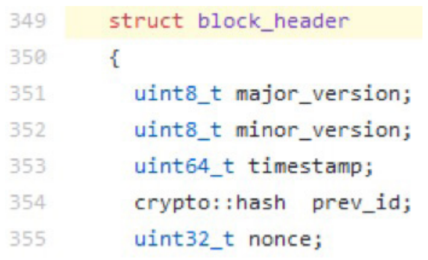
```
void SingleWorker::start()
{
    while (Workers::sequence() > 0) {
        if (Workers::isPaused()) {
            do {
                std::this_thread::sleep_for(std::chrono::milliseconds(200));
            }
            while (Workers::isPaused());

            if (Workers::sequence() == 0) {
                break;
            }

            consumeJob();
        }

        while (!Workers::isOutdated(m_sequence)) {
            if ((m_count & 0xF) == 0) {
                storeStats();
            }

            m_count++;
            *m_job.nonce() = ++m_result.nonce;

            if (CryptoNight::hash(m_job, m_result, m_ctx)) {
                Workers::submit(m_result);
            }

            std::this_thread::yield();
        }
```

*Figure 30*

While diving into the mechanics of the **CryptoNight::hash** method, it is important to consider:

1. The expected output of this function should be a yes/no status indicating whether or not the CryptoNight hash generated from the data blob in the Job object and our nonce produce a valid Monero PoW (determined by whether or not the hash is less than the target integer provided to us from the mining pool).

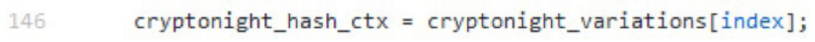
2. The data being hashed (the 96 byte data blob in the Job class declaration in Job.h) is the Monero block header (the final 32 bits of which are the nonce) plus the merkel root of all the TX which will be in the block plus the number of these TX. A good reference on the contents of the blob Monero mining pools distributed can be found here. In the Monero source code, the declaration of this block header concisely illuminates what the 76 bytes represent. Keep in mind that the final four bytes of this header (the nonce) do not need to be included in the data being sent to us by the mining pool since this is a value that we randomly generate and append on our own. Furthermore, the true size of the data being sent to us in the JSON object is actually half of what it appears to be (since it is sent in ASCII, where each hex byte has to be encoded as two characters). Ultimately, the data blob will consist of 75 fixed bytes plus nine variable bytes (for a maximum of 84) rounded to an alignment of 96. Figure 31 below shows the prototype of the block header structure.



```
struct block_header
{
  uint8_t major_version;
  uint8_t minor_version;
  uint64_t timestamp;
  crypto::hash  prev_id;
  uint32_t nonce;
```

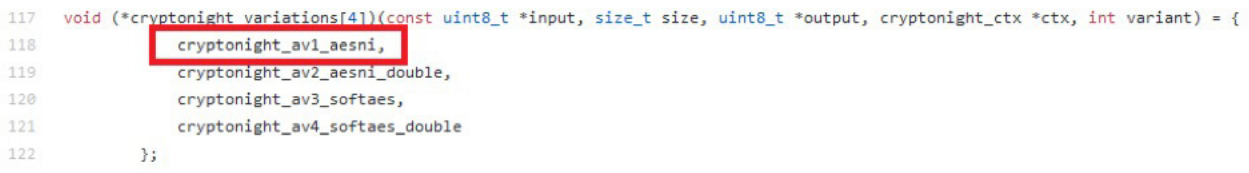
*Figure 31*

The **CryptoNight::hash** method is simply a wrapper function that selects the “correct” hashing function based upon the algorithm the miner is using (in our case, we are examining the standard CryptoNight AV1 AES algorithm). The specific hashing function for our specific algorithm was selected back when **CryptoNight::init** was called.



```
        cryptonight_hash_ctx = cryptonight_variations[index];
```

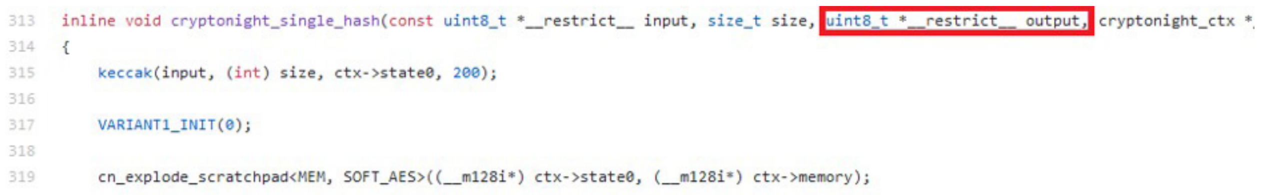
The hashing function pointer is stored in **cryptonight_hash_ctx** in **CryptoNight::init** in CryptoNight.cpp. It is selected by using the algorithm ID as an index into an array of hashing function pointers.



```
void (*cryptonight_variations[4])(const uint8_t *input, size_t size, uint8_t *output, cryptonight_ctx *ctx, int variant) = {
            cryptonight_av1_aesni,
            cryptonight_av2_aesni_double,
            cryptonight_av3_softaes,
            cryptonight_av4_softaes_double
        };
```

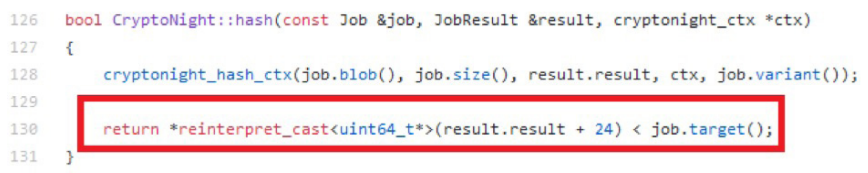
*Figure 32*

In our particular case, the function selected is **cryptonight_av1_aesni** (see the highlighted portion of Figure 32 above). This function will ultimately take us to the **cryptonight_single_hash** function in CryptoNight_x86.h, where the deep levels of technical implementation for CryptoNight are stored. The **cryptonight_single_hash** function does not directly return any value. Instead it stores its result within a 32 byte (256) output buffer meant to hold the resulting CryptoNight hash.



```
inline void cryptonight_single_hash(const uint8_t *__restrict__ input, size_t size, uint8_t *__restrict__ output, cryptonight_ctx *
{
    keccak(input, (int) size, ctx->state0, 200);

    VARIANT1_INIT(0);

    cn_explode_scratchpad<MEM, SOFT_AES>((__m128i*) ctx->state0, (__m128i*) ctx->memory);
```

From CryptoNight::hash, this output buffer is passed as a pointer to the result buffer field in the specified **JobResult** class object as shown in Figure 33 on the next page.

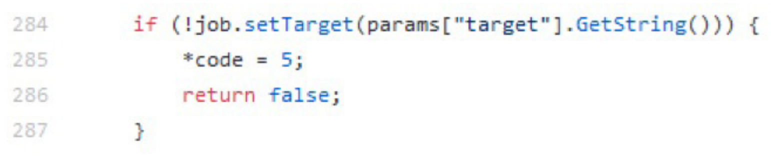


```
126 bool CryptoNight::hash(const Job &job, JobResult &result, cryptonight_ctx *ctx)
127 {
128     cryptonight_hash_ctx(job.blob(), job.size(), result.result, ctx, job.variant());
129
130     return *reinterpret_cast<uint64_t*>(result.result + 24) < job.target();
131 }
```

*Figure 33*

The **JobResult** class is simply a wrapper around a small block of 32 bytes (256 bits) of data in the form of a character array that allows this data to be manipulated and queried. These 256 bits of data represent the CryptoNight hash output for the data blob we passed in the Job object. The true/false return statement on line 130 (which will indicate to **SingleWorker::start** whether or not a valid nonce has been found to submit back to the mining pool) represents a characteristic of Monero pool mining distinctly different from solo mining.

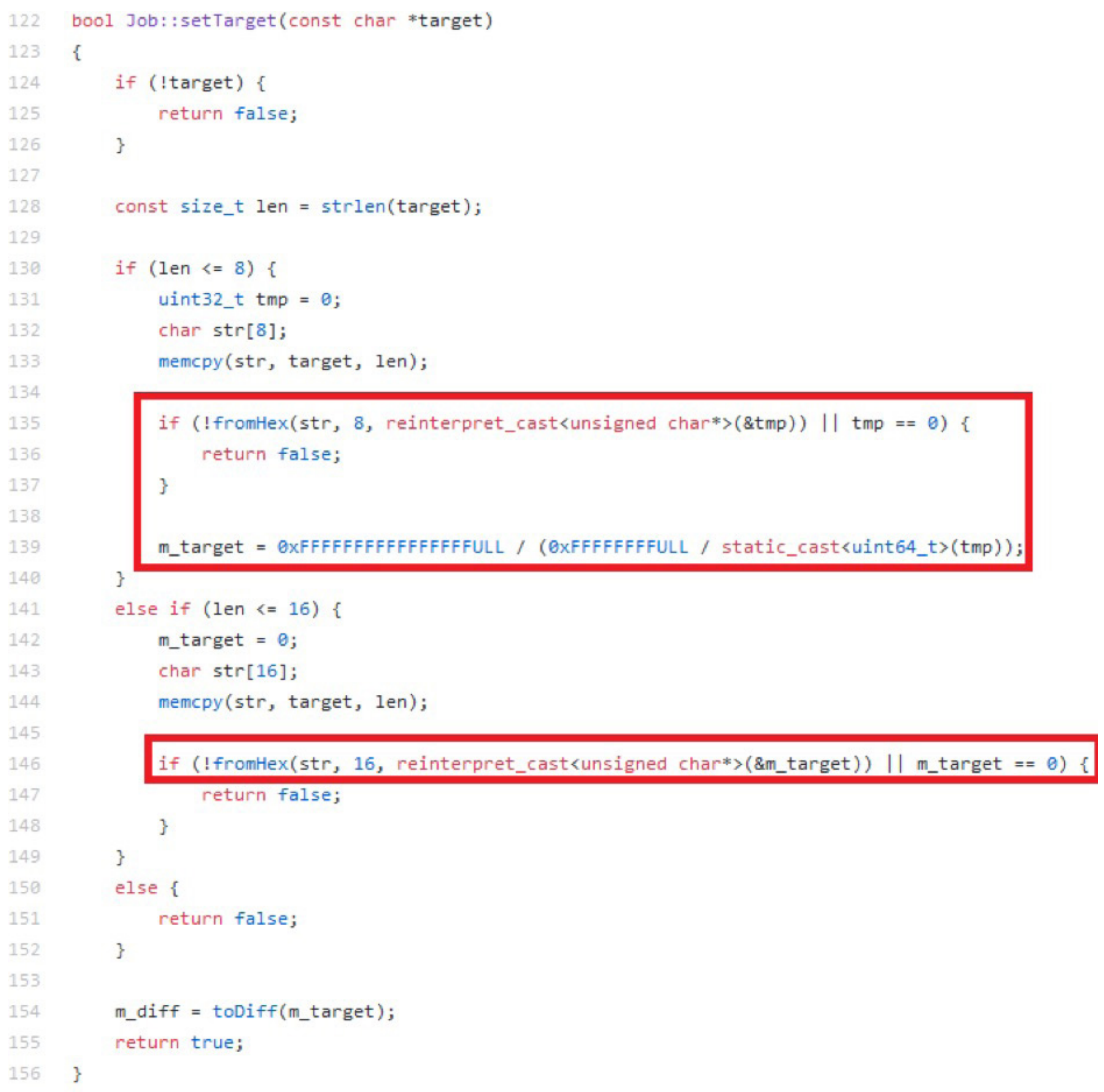
In Monero pool mining, a CryptoNight hash is considered a valid PoW for a block if it is “less than” a pre-defined “target” integer. In the XMRig implementation, we can see this logic at work on line 130 where the “.result” field (32 bytes of data representing the outputted hash) of the “result” class (a **JobResult** class object) is conditionally tested to be less than a “target” integer provided by the **Job::target** method (which, in turn, is simply returning the unsigned m_target integer value that was initialized in **Job::setTarget**, a method called using the “target” field of an incoming JSON document from the mining pool in Client.cpp). Figure 34 below shows this target value being parsed in string format from the received JSON data and passed to the setTarget method (further illustrated in Figure 35).



```
    if (!job.setTarget(params["target"].GetString())) {
        *code = 5;
        return false;
    }
```

*Figure 34*

The **Job::setTarget** method takes the ASCII encoded hex data within the “target” field of an incoming JSON job document from the mining pool (which can be either a 32- or 64-bit value, in the case of HashVault it is 32-bit) and converts it first to a raw byte array, and then to an unsigned 64-bit integer (later retrievable through the **Job::target** method).



```
bool Job::setTarget(const char *target)
{
    if (!target) {
        return false;
    }

    const size_t len = strlen(target);

    if (len <= 8) {
        uint32_t tmp = 0;
        char str[8];
        memcpy(str, target, len);

        if (!fromHex(str, 8, reinterpret_cast<unsigned char*>(&tmp)) || tmp == 0) {
            return false;
        }

        m_target = 0xFFFFFFFFFFFFFFFFULL / (0xFFFFFFFFULL / static_cast<uint64_t>(tmp));
    }
    else if (len <= 16) {
        m_target = 0;
        char str[16];
        memcpy(str, target, len);

        if (!fromHex(str, 16, reinterpret_cast<unsigned char*>(&m_target)) || m_target == 0) {
            return false;
        }
    }
    else {
        return false;
    }

    m_diff = toDiff(m_target);
    return true;
}
```

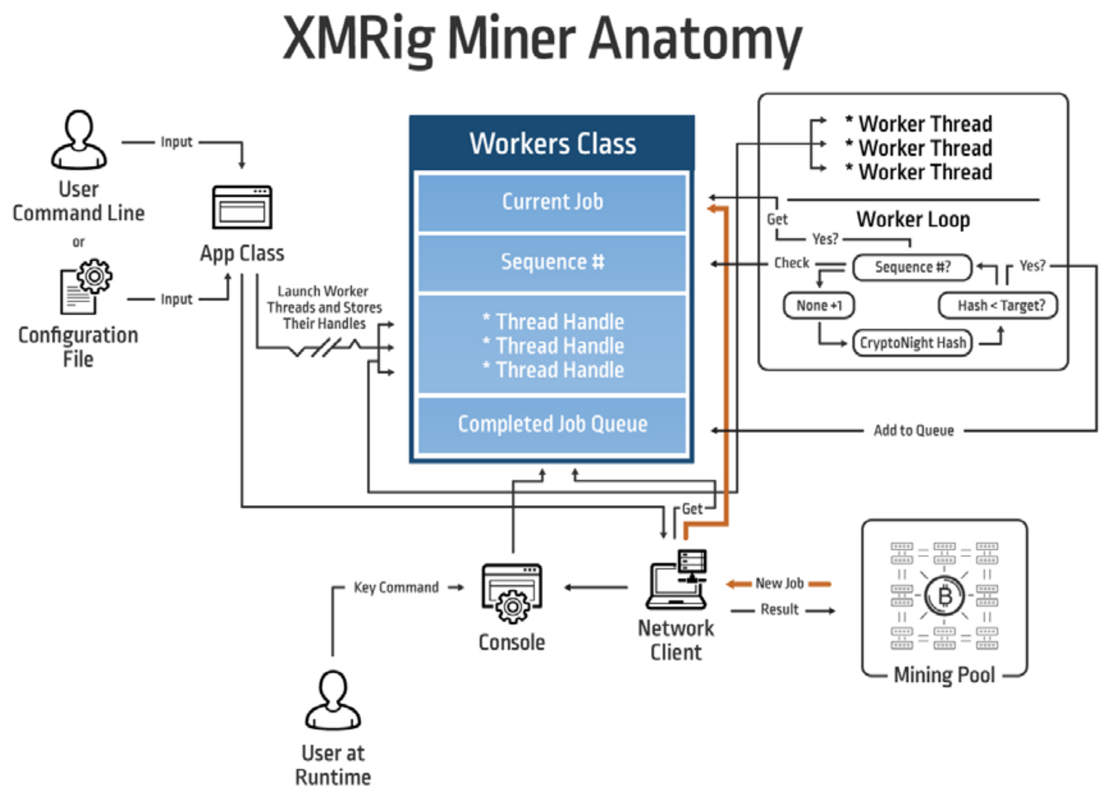
*Figure 35*

It's important to note that the target value is determined by the pool itself, not the blockchain. While the pool miner may be solving a genuine block, he/she does not need to solve the block in such a way that would add it to the blockchain in order to receive a payout from the mining pool. Instead, the mining pool's payouts are based on the amount of computing power the miner contributes to solving each block. This method works under the assumption that eventually one of the pool miners will get lucky and submit a hash that satisfies the (much more difficult) criteria required to solve a block, which the pool can then add to the blockchain.

While the full implementation details of the CryptoNight algorithm are much too complex to describe here, a good resource on the topic can be found here.

## XMRig Takeaways

XMRig is a well-written and modular cryptocurrency miner. It has a low dependence on external libraries and is easily constructed even by novice programmers. It is an ideal candidate for a malicious cryptocurrency miner because of its small size, fast speed, ease of use and Windows support.



XMRig works by initiating a connection to a mining pool, then receiving jobs in the form of data blobs (representing potential Monero headers) and target values (to determine the success of a PoW using CryptoNight). These jobs are then placed in an intermediary global class called Workers, where they are consumed by a variable number of worker threads (synchronized via sequence number), all working to calculate a hash that has a lower value than the target value by repeatedly incrementing a 32-bit nonce value. When a miner thread succeeds in calculating a valid PoW, it sends its resulting hash and nonce value back to the mining pool where it will be accepted and result in a credit to the user's Monero wallet address.

## Detection Vectors

While our analysis of the XMRig source code has provided insight into how a malicious cryptominer running on Windows would work in practice, it admittedly provides very little in the way of detection vectors. The vast majority of the code is much too generic in nature to provide any unique weaknesses to target with security software.

Instead, security software can be leveraged to pinpoint unique or abnormal behavioral patterns, such as specific API call/ETW event sequences to indicate the presence of malware. A classic example would be the use of process enumeration and then injection into a web browser process for a banking trojan, or a widespread file enumeration in conjunction with file I/O and cryptographic API usage for ransomware.

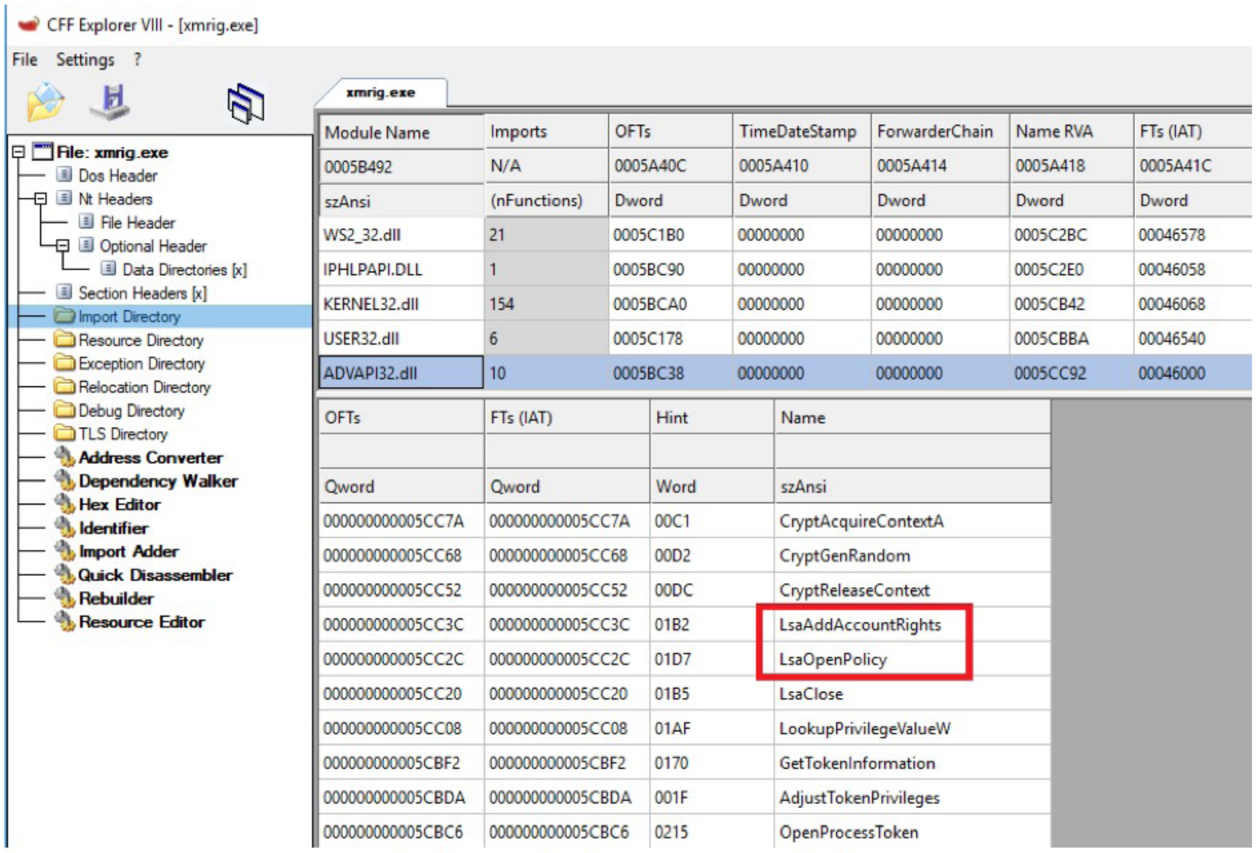
For the emerging genre of cryptominers, much fewer useful indicators exist, and for those that do, the risk of false positives could prove a formidable obstacle. The design approach taken in XMRig could vary to a very high degree between cryptominers, however, there are a few small points of interest we should examine for potential detection of this new malware genre.

## Huge Page Allocation

One of the highly unique characteristics of Monero mining is its use of the CryptoNight algorithm to enforce a computing power bottleneck through memory latency (via its reliance on 2MB of scratchpad memory to calculate hashes). As discussed earlier, this bottleneck puts memory access speed at the forefront with hash computation, and optimizations to memory access speed can increase miner output dramatically.

In Windows, the VirtualAlloc API provides a unique method of memory latency optimization through its MEM_LARGE_PAGES flag. This flag could prove to be a highly useful indicator while hooking VirtualAlloc to detect the initialization of a Monero miner.

Additionally, the unique user account privileges required to utilize the MEM_LARGE_PAGES flag also provide a useful (albeit one-off) means of detecting a Monero miner through API hooking/ETW event consumption. If the malware writer fails to obfuscate his/her miner properly, the visibility of the Windows API calls needed to modify these privileges could provide an excellent indication of a Monero miner through the import section in the PE image (unlike VirtualAlloc, which is far too commonly found in import sections to be used as an indicator here). Figure 36 below illustrates the contents of the import section in the XMRig PE file through a tool called CFF Explorer.

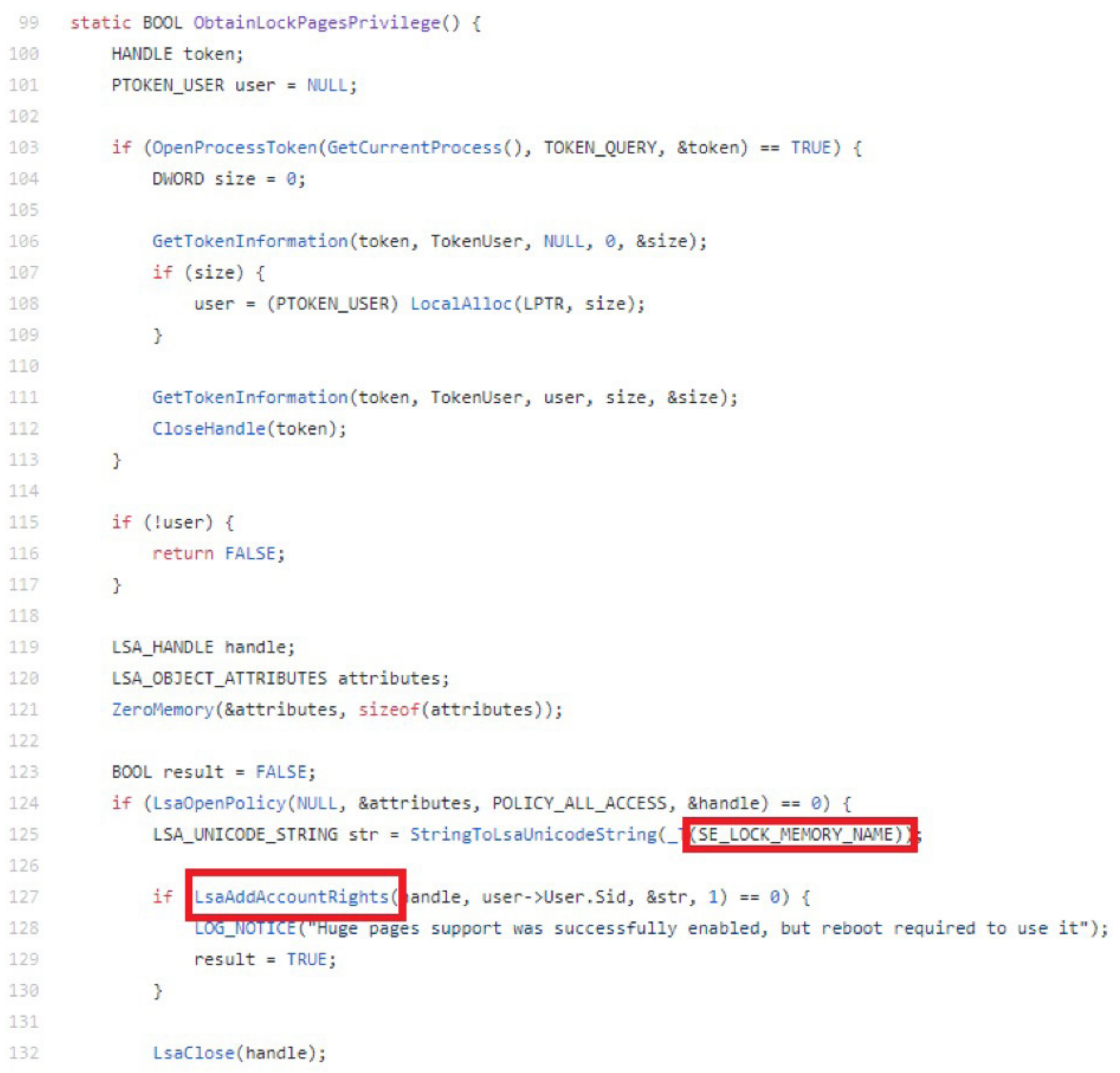


| Module Name | Imports | OFTs | TimeDateStamp | ForwarderChain | Name RVA | FTs (IAT) |
|---|---|---|---|---|---|---|
| 0005B492 | N/A | 0005A40C | 0005A410 | 0005A414 | 0005A418 | 0005A41C |
| szAnsi | (nFunctions) | Dword | Dword | Dword | Dword | Dword |
| WS2_32.dll | 21 | 0005C1B0 | 00000000 | 00000000 | 0005C2BC | 00046578 |
| IPHLPAPI.DLL | 1 | 0005BC90 | 00000000 | 00000000 | 0005C2E0 | 00046058 |
| KERNEL32.dll | 154 | 0005BCA0 | 00000000 | 00000000 | 0005CB42 | 00046068 |
| USER32.dll | 6 | 0005C178 | 00000000 | 00000000 | 0005CBBA | 00046540 |
| ADVAPI32.dll | 10 | 0005BC38 | 00000000 | 00000000 | 0005CC92 | 00046000 |

| OFTs | FTs (IAT) | Hint | Name |
|---|---|---|---|
| | | | |
| Qword | Qword | Word | szAnsi |
| 000000000005CC7A | 000000000005CC7A | 00C1 | CryptAcquireContextA |
| 000000000005CC68 | 000000000005CC68 | 00D2 | CryptGenRandom |
| 000000000005CC52 | 000000000005CC52 | 00DC | CryptReleaseContext |
| 000000000005CC3C | 000000000005CC3C | 01B2 | LsaAddAccountRights |
| 000000000005CC2C | 000000000005CC2C | 01D7 | LsaOpenPolicy |
| 000000000005CC20 | 000000000005CC20 | 01B5 | LsaClose |
| 000000000005CC08 | 000000000005CC08 | 01AF | LookupPrivilegeValueW |
| 000000000005CBF2 | 000000000005CBF2 | 0170 | GetTokenInformation |
| 000000000005CBDA | 000000000005CBDA | 001F | AdjustTokenPrivileges |
| 000000000005CBC6 | 000000000005CBC6 | 0215 | OpenProcessToken |

*Figure 36*

Note that the APIs in the import section of a compiled XMRig binary above correspond to the privilege modification code described earlier in this paper (see Figure 37 on the next page).

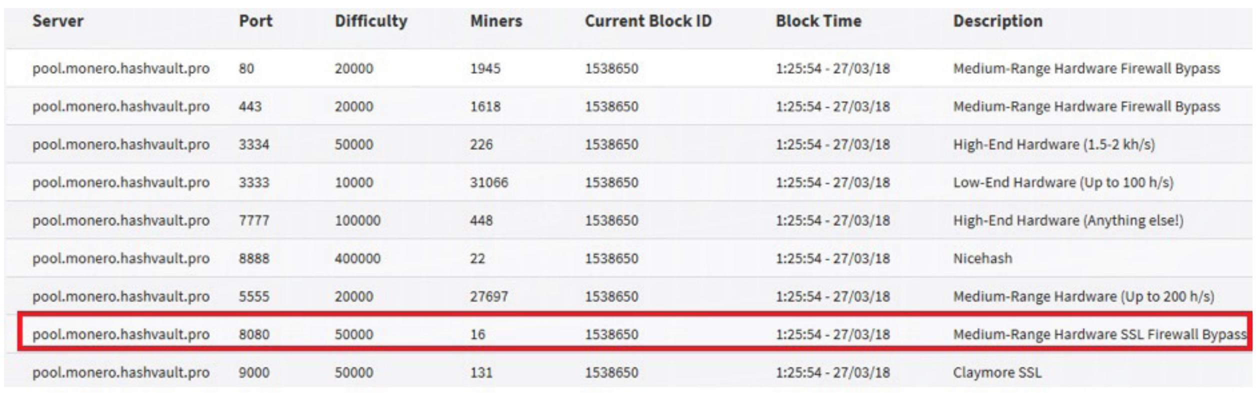


```
static BOOL ObtainLockPagesPrivilege() {
    HANDLE token;
    PTOKEN_USER user = NULL;

    if (OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &token) == TRUE) {
        DWORD size = 0;

        GetTokenInformation(token, TokenUser, NULL, 0, &size);
        if (size) {
            user = (PTOKEN_USER) LocalAlloc(LPTR, size);
        }

        GetTokenInformation(token, TokenUser, user, size, &size);
        CloseHandle(token);
    }

    if (!user) {
        return FALSE;
    }

    LSA_HANDLE handle;
    LSA_OBJECT_ATTRIBUTES attributes;
    ZeroMemory(&attributes, sizeof(attributes));

    BOOL result = FALSE;
    if (LsaOpenPolicy(NULL, &attributes, POLICY_ALL_ACCESS, &handle) == 0) {
        LSA_UNICODE_STRING str = StringToLsaUnicodeString(_T(SE_LOCK_MEMORY_NAME));

        if (LsaAddAccountRights(handle, user->User.Sid, &str, 1) == 0) {
            LOG_NOTICE("Huge pages support was successfully enabled, but reboot required to use it");
            result = TRUE;
        }

        LsaClose(handle);
```

*Figure 37*

## Mining Pool Network Traffic

While the outbound traffic produced by a malicious cryptominer would be an excellent indication of infection, there are two challenges to consider: the ports are inconsistent between different mining pools, and some also have SSL encryption available to mask the underlying traffic. Take, for example, a list of available Monero mining service/ports listed on the HashVault website (see Figure 38 below).



| Server | Port | Difficulty | Miners | Current Block ID | Block Time | Description |
|---|---|---|---|---|---|---|
| pool.monero.hashvault.pro | 80 | 20000 | 1945 | 1538650 | 1:25:54 - 27/03/18 | Medium-Range Hardware Firewall Bypass |
| pool.monero.hashvault.pro | 443 | 20000 | 1618 | 1538650 | 1:25:54 - 27/03/18 | Medium-Range Hardware Firewall Bypass |
| pool.monero.hashvault.pro | 3334 | 50000 | 226 | 1538650 | 1:25:54 - 27/03/18 | High-End Hardware (1.5-2 kh/s) |
| pool.monero.hashvault.pro | 3333 | 10000 | 31066 | 1538650 | 1:25:54 - 27/03/18 | Low-End Hardware (Up to 100 h/s) |
| pool.monero.hashvault.pro | 7777 | 100000 | 448 | 1538650 | 1:25:54 - 27/03/18 | High-End Hardware (Anything else!) |
| pool.monero.hashvault.pro | 8888 | 400000 | 22 | 1538650 | 1:25:54 - 27/03/18 | Nicehash |
| pool.monero.hashvault.pro | 5555 | 20000 | 27697 | 1538650 | 1:25:54 - 27/03/18 | Medium-Range Hardware (Up to 200 h/s) |
| pool.monero.hashvault.pro | 8080 | 50000 | 16 | 1538650 | 1:25:54 - 27/03/18 | Medium-Range Hardware SSL Firewall Bypass |
| pool.monero.hashvault.pro | 9000 | 50000 | 131 | 1538650 | 1:25:54 - 27/03/18 | Claymore SSL |

*Figure 38*

If the malware author has failed to properly obscure their cryptominer traffic, either through the use of SSL and/or a proxy, these attributes provide the simplest and most valuable indications of a cryptominer infection on both a local machine and network level. Since there is a small and finite number of mining pools, connections to the IPs of these pools are a dead giveaway of an infection (DNS is less effective since the malware author could register his/her own domains to point to the pool IPs in order to avoid directly resolving host names such as "pool.monero.hashvault.pro"). If the destination IP and host name are obscured through the use of a proxy, traffic patterns between the pool and miner (passing uniquely crafted JSON documents back and forth) could be another good indicator, albeit less practical and easily bypassed through the use of SSL by an attacker.

### CryptoNight Logic

While the lack of Windows API calls within the CryptoNight logic itself makes it invisible from the point of view of an API hook or ETW event consumer, it provides a powerful/reliable method of detection for traditional byte pattern file scanners in security products such as antivirus. The reason for this is that it utilizes highly specific code patterns, which are:

- Essential for the miner to function. Without its own CryptoNight functionality, a Monero miner is not going to be able to achieve his/her basic objectives.
- Relatively immutable since without an advanced understanding of the cryptographic logic (a rare expertise), an attacker is not going to be able to make major changes to the code without breaking it.
- Attractive for copy/paste. The amount of effort required to create a new CryptoNight implementation from scratch far exceeds the reward from an attacker's perspective. The attackers who even bother to create custom miners (as opposed to those who just run a pre-compiled miner EXE file with their own config) are very likely to copy files such as CryptoNight_x86.h from projects such as XMRig directly into their own code base to save time.

In addition to depending on pre-written code in the form of the CryptoNight algorithm itself, CryptoNight has additional cryptographic dependencies in the form of pre-written code from:

- Blake256 (see c_blake256.c)
- Groestl (see c_groestl.c)
- JH (see c_jh.c)
- Keccak (see c_keccak.c)
- Skein (see c_skein.c)

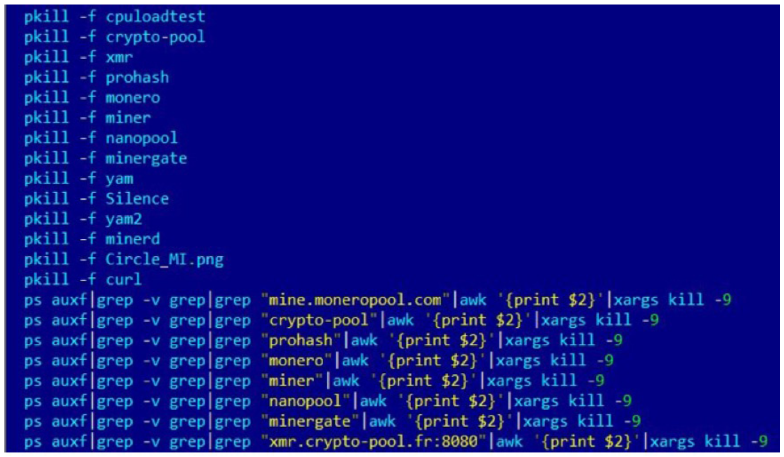
These algorithms carry the same issues as CryptoNight from an attacker's perspective: very high complexity (immutable without a background in cryptography) and inundation with unique code patterns and data tables.

### User-Facing Strings and Command Line

Another easy infection vector for cryptominers in general (and XMRig is no exception) is that there is a great abundance of human readable strings, oftentimes unique ones, which can make strong byte pattern signatures on their own. Since all of these public miners were written with user interaction in mind (rather than stealth from security products), there are large amounts of user interaction code in the form of strings embedded in them, which require removal skills that an amateur attacker may not possess. The solution to this issue of unique strings from an attacker's point of view falls along the lines of his/her skill level.

- For low-skill attackers, simply packing the miner binary so that it will not touch disk may be sufficient.
- For medium-skill level attackers, programmatically removing all human readable strings (or encrypting the ones that cannot be removed, such as JSON strings) and user interaction code in the miner and then recompiling it may be enough.
- For advanced attackers, writing a miner from scratch with stealth in mind would yield the best solution.

As might be expected, an even simpler variation on this detection technique can be performed without scanning a file's contents by simply watching for suspicious command lines on active processes. Since these command line syntaxes aren't going to change without directly modifying/recompiling a miner, they can provide a reliable infection vector on their own. A cryptominer malware from October 2017 even went so far as to use this concept as a generic detection technique to kill competing cryptominers on machines it compromised by scanning active process command lines for mining pool domain strings (see Figure 39 on the next page for an illustration of this approach taken from an actual security product).



```
pkill -f cpuloadtest
pkill -f crypto-pool
pkill -f xmr
pkill -f prohash
pkill -f monero
pkill -f miner
pkill -f nanopool
pkill -f minergate
pkill -f yam
pkill -f Silence
pkill -f yam2
pkill -f minerd
pkill -f Circle_MI.png
pkill -f curl
ps auxf|grep -v grep|grep "mine.moneropool.com"|awk '{print $2}'|xargs kill -9
ps auxf|grep -v grep|grep "crypto-pool"|awk '{print $2}'|xargs kill -9
ps auxf|grep -v grep|grep "prohash"|awk '{print $2}'|xargs kill -9
ps auxf|grep -v grep|grep "monero"|awk '{print $2}'|xargs kill -9
ps auxf|grep -v grep|grep "miner"|awk '{print $2}'|xargs kill -9
ps auxf|grep -v grep|grep "nanopool"|awk '{print $2}'|xargs kill -9
ps auxf|grep -v grep|grep "minergate"|awk '{print $2}'|xargs kill -9
ps auxf|grep -v grep|grep "xmr.crypto-pool.fr:8080"|awk '{print $2}'|xargs kill -9
```

*Figure 39*

A similar technique was also taken in a free cryptominer removal script written by Minerva Labs, which searches for strings such as "cryptonight" and various mining pool URLs in active process command lines. This approach has become increasingly common. Notably, it was used in the recent fileless GhostMiner malicious cryptominer.

### CPU Usage

The heavy CPU usage, spawning of multiple threads and manipulation of thread/process CPU priority level via the SetPriorityClass and SetThreadPriority APIs are all good indicators of a cryptominer, but are far too prone to false positives to be used on their own for detection. At best, these types of behaviors could be combined with other detection criteria (such as recognizable command line strings) to increase detection accuracy.

## Conclusion

Monero cryptominers will increasingly pose a unique challenge to security teams and products, as they do not follow historic malware principles. Unlike any other genre of malware, cryptominers don't need to "interact" with the OS by logging keystrokes, making process injections, taking screenshots, encrypting files or raising flags through other noisy, high-impact behavior. Instead, they need only hash data quietly, in the background.

Another unique quality of this emerging malware genre is the polarization of detection complexity based on the skill level of the attacker. A cryptominer used by an amateur attacker may be among the easiest types of malware to detect (even if it is unknown) since it is likely to use pre-compiled miner executables and large amounts of recycled cryptographic code, while depending heavily on unmasked connections to mining pools (for which a blacklist would be trivial to create). Additionally, this malware is more likely to be identified by the average user (with ransomware being a notable exception) since it must drastically degrade system performance to achieve its purpose.

Conversely, when designed by an advanced attacker, cryptomining malware may prove to be one of the most difficult genres of malware to detect, due to its minimal Windows API usage, low user interaction and input levels and small list of requirements needed to carry out its mission. All it takes is a connection to a mining pool, the ability to automatically run when the system starts and the capacity to consume a large number of CPU cycles.

For this reason, we can expect cryptominers—and the anonymous Monero cryptocurrency it targets—to remain a ubiquitous feature of the constantly evolving threat landscape for years to come.

**Author: Forrest Williams**

©Copyright 1999-2018 CyberArk Software. All rights reserved. No portion of this publication may be reproduced in any form or by any means without the express written consent of CyberArk Software. CyberArk®, the CyberArk logo and other trade or service names appearing above are registered trademarks (or trademarks) of CyberArk Software in the U.S. and other jurisdictions. Any other trade and service names are the property of their respective owners. U.S., 01.18. 205301743

CyberArk believes the information in this document is accurate as of its publication date. The information is provided without any express, statutory, or implied warranties and is subject to change without notice.

THIS PUBLICATION IS FOR INFORMATIONAL PURPOSES ONLY AND IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER WHETHER EXPRESSED OR IMPLIED, INCLUDING WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, NON-INFRINGEMENT OR OTHERWISE. IN NO EVENT SHALL CYBERARK BE LIABLE FOR ANY DAMAGES WHATSOEVER, AND IN PARTICULAR CYBERARK SHALL NOT BE LIABLE FOR DIRECT, SPECIAL, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES, OR DAMAGES FOR LOST PROFITS, LOSS OF REVENUE OR LOSS OF USE, COST OF REPLACEMENT GOODS, LOSS OR DAMAGE TO DATA ARISING FROM USE OF OR IN RELIANCE ON THIS PUBLICATION, EVEN IF CYBERARK HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.