Cado Discovers Denonia: The First Malware Specifically Targeting Lambda

: 4/6/2022

CADO//
Blog
Cado Discovers Denonia: The First Malware Specifically Targeting Lambda

April 6, 2022

By Matt Muir, with thanks to Chris Doman, Al Carchrie and Paul Scott.

Organisations – both large and small – are increasingly leveraging Lambda serverless functions. From a business agility perspective, serverless has significant benefits. Lambda also brings security benefits – the managed runtime environment reduces the attack surface compared to a more traditional server environment. However, short runtime durations, the sheer volume of executions, and the dynamic and ephemeral nature of Lambda functions can make it difficult to detect, investigate and respond to a potential compromise. Under the AWS Shared Responsibility model, AWS secures the underlying Lambda execution environment but it is up to the customer to secure functions themselves.

Cado Labs routinely analyses cloud environments to look for the latest threats. As part of ongoing research, we found the first publicly-known case of malware specifically designed to execute in an AWS Lambda environment. We named this malware Denonia, after the name the attackers gave the domain it communicates with. The malware uses newer address resolution techniques for command and control traffic to evade typical detection measures and virtual network access controls. Although this first sample is fairly innocuous in that it only runs crypto-mining software, it demonstrates how attackers are using advanced cloud-specific knowledge to exploit complex cloud infrastructure, and is indicative of potential future, more nefarious attacks. From the telemetry we have seen, the distribution of Denonia so far has been limited.

Technical Analysis

We initially came across a sample of Denonia with the following SHA-256 hash:

- a31ae5b7968056d8d99b1b720a66a9a1aeee3637b97050d95d96ef3a265cbbca

Whilst it has the filename "python" – the malware is actually written in Go and seems to contain a customised variant of the XMRig mining software, along with other unknown functions. We decided to investigate further.

During dynamic analysis, the malware quickly halted execution and logged the following error:

```
2022/04/01 11:37:21 expected AWS Lambda environment variables
[_LAMBDA_SERVER_PORT AWS_LAMBDA_RUNTIME_API] are not defined
```

This piqued our interest as these environment variables are specific to Lambda, giving us some hints about the environment in which this malware is expected to execute. Reviewing the binary further, we could see that it was a 64-bit ELF executable targeting the x86-64 architecture and that it uses a number of third-party libraries, including one specifically to enable execution inside AWS Lambda environments.

A Note on Go Malware

Malware written in Google's Go programming language has become increasingly prevalent in recent years. The language is attractive to malware developers for a number of reasons, including the ease in which it can produce cross-compatible executables and the efficient deployment that statically-linked binaries bring. However, these characteristics of the language can pose some challenges to malware researchers analysing binaries compiled from Go.

Firstly, statically-linked binaries are typically much larger than dynamically-linked equivalents – this makes static analysis slightly more laborious. Go also handles strings in an unusual way. Strings are not null-terminated, as they are in C-like languages, instead they are stored in a large blob and a struct which includes both a pointer to the string in the blob and an integer defining its length is created upon declaration. This can confuse some static analysis tools.

Analysing Lambda Malware

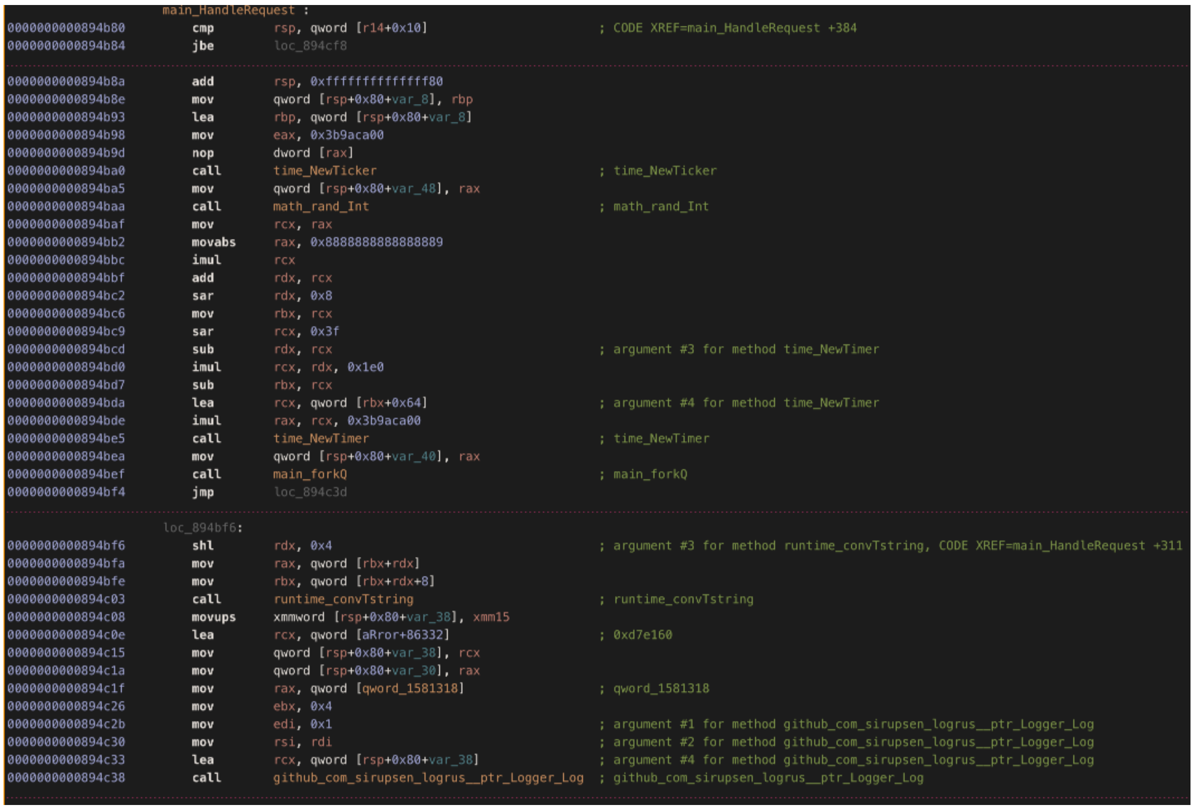
Analysing a binary designed to run in AWS Lambda poses some interesting challenges.

Whilst Denonia is clearly designed to execute inside of Lambda environments – we haven't yet identified how it is deployed. It may simply be a matter of compromising AWS Access and Secret Keys then manually deploying into compromised Lambda environments, as we've seen before with more simple Python scripts.

Using the redress tool we identified some interesting third-party Go libraries that the malware embeds. This gave us some clues about its functionality:

- **github.com/aws/aws-lambda-go/lambda** – libraries, samples and tools for writing Lambda functions in Go
- **github.com/aws/aws-lambda-go/lambdacontext** – helpers for retrieving contextual information from a Lambda invoke request
- **github.com/aws/aws-sdk-go/aws** – general AWS SDK for Golang
- **github.com/likexian/doh-go** – DNS over HTTPS in Go, supports providers such as Quad9, Cloudflare etc

We can see a snippet of the Lambda function handler below, which expects certain data to be set:



```
                   main_HandleRequest :
0000000000894b80        cmp        rsp, qword [r14+0x10]                ; CODE XREF=main_HandleRequest +384
0000000000894b84        jbe        loc_894cf8

0000000000894b8a        add        rsp, 0xffffffffffffff80
0000000000894b8e        mov        qword [rsp+0x80+var_8], rbp
0000000000894b93        lea        rbp, qword [rsp+0x80+var_8]
0000000000894b98        mov        eax, 0x3b9aca00
0000000000894b9d        nop        dword [rax]
0000000000894ba0        call       time_NewTicker                       ; time_NewTicker
0000000000894ba5        mov        qword [rsp+0x80+var_48], rax
0000000000894baa        call       math_rand_Int                        ; math_rand_Int
0000000000894baf        mov        rcx, rax
0000000000894bb2        movabs     rax, 0x8888888888888889
0000000000894bbc        imul       rcx
0000000000894bbf        add        rdx, rcx
0000000000894bc2        sar        rdx, 0x8
0000000000894bc6        mov        rbx, rcx
0000000000894bc9        sar        rcx, 0x3f
0000000000894bcd        sub        rdx, rcx                             ; argument #3 for method time_NewTimer
0000000000894bd0        imul       rcx, rdx, 0x1e0
0000000000894bd7        sub        rbx, rcx
0000000000894bda        lea        rcx, qword [rbx+0x64]                ; argument #4 for method time_NewTimer
0000000000894bde        imul       rax, rcx, 0x3b9aca00
0000000000894be5        call       time_NewTimer                        ; time_NewTimer
0000000000894bea        mov        qword [rsp+0x80+var_40], rax
0000000000894bef        call       main_forkQ                           ; main_forkQ
0000000000894bf4        jmp        loc_894c3d

                   loc_894bf6:
0000000000894bf6        shl        rdx, 0x4                             ; argument #3 for method runtime_convTstring, CODE XREF=main_HandleRequest +311
0000000000894bfa        mov        rax, qword [rbx+rdx]
0000000000894bfe        mov        rbx, qword [rbx+rdx+8]
0000000000894c03        call       runtime_convTstring                  ; runtime_convTstring
0000000000894c08        movups     xmmword [rsp+0x80+var_38], xmm15
0000000000894c0e        lea        rcx, qword [aRror+86332]             ; 0xd7e160
0000000000894c15        mov        qword [rsp+0x80+var_38], rcx
0000000000894c1a        mov        qword [rsp+0x80+var_30], rax
0000000000894c1f        mov        rax, qword [qword_1581318]           ; qword_1581318
0000000000894c26        mov        ebx, 0x4
0000000000894c2b        mov        edi, 0x1                             ; argument #1 for method github_com_sirupsen_logrus__ptr_Logger_Log
0000000000894c30        mov        rsi, rdi                             ; argument #2 for method github_com_sirupsen_logrus__ptr_Logger_Log
0000000000894c33        lea        rcx, qword [rsp+0x80+var_38]         ; argument #4 for method github_com_sirupsen_logrus__ptr_Logger_Log
0000000000894c38        call       github_com_sirupsen_logrus__ptr_Logger_Log ; github_com_sirupsen_logrus__ptr_Logger_Log
```

Despite the presence of this, we discovered during dynamic analysis that the sample will happily continue execution outside a Lambda environment (i.e. on a vanilla Amazon Linux box). We suspect this is likely due to Lambda "serverless" environments using Linux under the hood, so the malware believed it was being run in Lambda (after we manually set the required environment variables) despite being run in our sandbox.

DNS over HTTPS

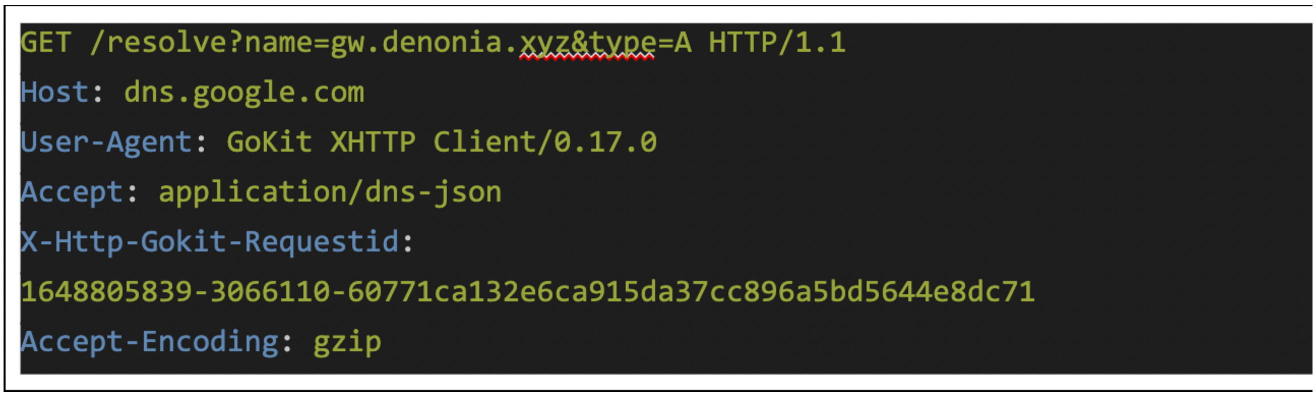
Normally when you request a domain name such as google.com, you send out an unencrypted DNS request to find the IP Address the domain resolves to – so your machine can connect to the domain. A relatively modern replacement for traditional DNS is DNS over HTTPS (DoH). DoH encrypts DNS queries, and sends the requests out as regular HTTPS traffic to DoH resolvers.

Using DoH is a fairly unusual choice for the Denonia authors, but provides two advantages here:

- AWS cannot see the dns lookups for the malicious domain, reducing the likelihood of triggering a detection
- Some Lambda environments may be unable to perform DNS lookups, depending on VPC settings.

We can see the malware sending these requests using the "doh-go" library to URLs such as:

- https://cloudflare-dns.com/dns-query?name=gw.denonia.xyz&type=A
- https://dns.google.com/resolve?name=gw.denonia.xyz&type=A



```
GET /resolve?name=gw.denonia.xyz&type=A HTTP/1.1
Host: dns.google.com
User-Agent: GoKit XHTTP Client/0.17.0
Accept: application/dns-json
X-Http-Gokit-Requestid:
1648805839-3066110-60771ca132e6ca915da37cc896a5bd5644e8dc71
Accept-Encoding: gzip
```

The HTTPS Request to the Google DoH Server

And the DoH server (in this case from Google) responds with the IP the domain resolves to in a JSON format:

```
{
    "Status": 0, "TC": false, "RD": true, "RA": true, "AD": false, "CD":
false,
    "Question": [{ "name": "gw.denonia.xyz.", "type": 1
    }],
    "Answer": [{ "name": "gw.denonia.xyz.",
        "type": 1, "TTL": 60,
        "data": "116.203.4.0"
    }],
    "Comment": "Response from 88.198.229.192."
}
```

Writing this to /tmp/.xmrig.json for XMRig

The attacker controlled domain gw.denonia[.]xyz resolves to 116.203.4[.]0 – which is then written into a config file for xmrig at /tmp/.xmrig.json:

```
"pools": [
    {
        "url": "116.203.4.0:3333",
        "user": "echonet.amd64",
        "pass": null,
        "rig-id": "echonet.amd64"
```

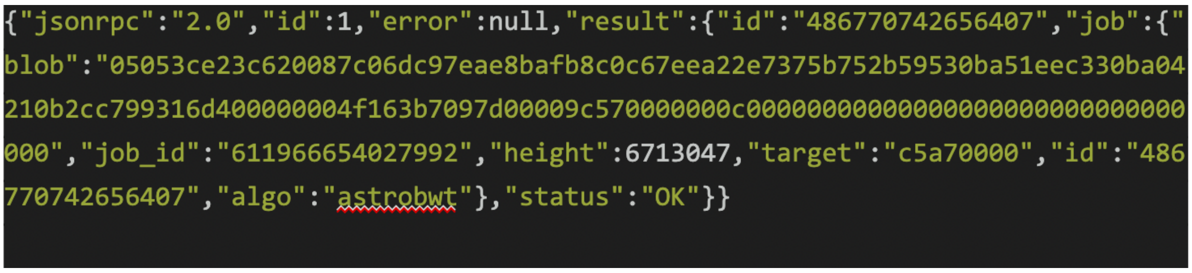
Note that on AWS Lambda, the only directory that you can write to is /tmp. The binary also sets the HOME directory to /tmp with "HOME=/tmp". XMRig itself is executed from memory.

Communication with the Monero server at 116.203.4[.]0

Denonia then starts XMRig from memory, and communicates with the attacker controlled Mining pool at 116.203.4[.]0:3333:

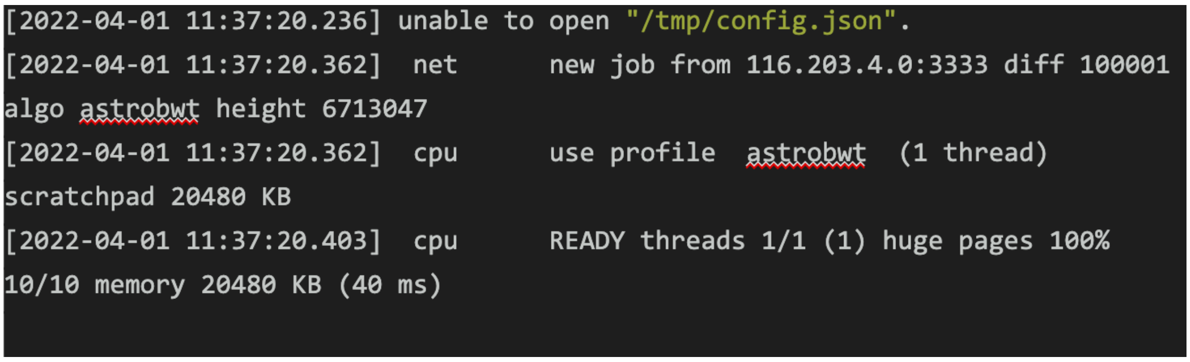
```
{"id":1,"jsonrpc":"2.0","method":"login","params":{"login":"echonet.amd64","pa
ss":"x","agent":"XMRig/6.15.2 (Linux x86_64) libuv/1.42.0
gcc/10.3.1","rigid":"echonet.amd64","algo":["cn/1","cn/2","cn/r","cn/fast","cn
/half","cn/xao","cn/rto","cn/rwz","cn/zls","cn/double","cn/ccx","cn-lite/1","c
n-heavy/0","cn-heavy/tube","cn-heavy/xhv","cn-pico","cn-pico/tlo","cn/upx2","r
x/0","rx/wow","rx/arq","rx/graft","rx/sfx","rx/keva","argon2/chukwa","argon2/c
hukwav2","argon2/ninja","astrobwt"]}}
```

Which replies with the status of the mining job:



```
{"jsonrpc":"2.0","id":1,"error":null,"result":{"id":"486770742656407","job":{"
blob":"05053ce23c620087c06dc97eae8bafb8c0c67eea22e7375b752b59530ba51eec330ba04
210b2cc799316d400000004f163b7097d00009c570000000c00000000000000000000000000000
000","job_id":"611966654027992","height":6713047,"target":"c5a70000","id":"486
770742656407","algo":"astrobwt"},"status":"OK"}}
```

XMRig also writes to the console as it executes:



```
[2022-04-01 11:37:20.236] unable to open "/tmp/config.json".
[2022-04-01 11:37:20.362]  net      new job from 116.203.4.0:3333 diff 100001
algo astrobwt height 6713047
[2022-04-01 11:37:20.362]  cpu      use profile  astrobwt  (1 thread)
scratchpad 20480 KB
[2022-04-01 11:37:20.403]  cpu      READY threads 1/1 (1) huge pages 100%
10/10 memory 20480 KB (40 ms)
```

More Malware Samples

Interestingly – this isn't the only sample of Denonia. Whilst the first sample we looked at dates from the end of February, we also found a second sample that was uploaded to VirusTotal in January 2022:

- 739fe13697bc55870ceb35003c4ee01a335f9c1f6549acb6472c5c3078417eed

Stay tuned for additional blogs on Lambda malware!

Investigating AWS Lambda Environments

This week we added the ability to investigate and remediate both AWS ECS and AWS Lambda environments to Cado Response. You can get a free playbook on how to investigate and respond to compromises in ECS here, and a free trial of the platform itself here.

For more on securing AWS Lambda environments, see the Whitepaper from AWS.

Indicators of Compromise

```
rule lambda_malware
{
    meta:
        description = "Detects AWS Lambda Malware"
        author = "[email protected]"
        license = "Apache License 2.0"
        date = "2022-04-03"
        hash1 = "739fe13697bc55870ceb35003c4ee01a335f9c1f6549acb6472c5c3078417eed"
        hash2 = "a31ae5b7968056d8d99b1b720a66a9a1aeee3637b97050d95d96ef3a265cbbca"
    strings:
        $a = "github.com/likexian/doh-go/provider/"
        $b = "Mozilla/5.0 (compatible; Ezooms/1.0; [email protected])"
        $c = "username:password pair for mining server"
    condition:
        filesize < 30000KB and all of them
}
```

IOCs

SHA256

739fe13697bc55870ceb35003c4ee01a335f9c1f6549acb6472c5c3078417eed
a31ae5b7968056d8d99b1b720a66a9a1aeee3637b97050d95d96ef3a265cbbca

Domains

denonia[.]xyz
ctrl.denonia[.]xyz
gw.denonia[.]xyz
1.gw.denonia[.]xyz
www.denonia[.]xyz
xyz.denonia[.]xyz
mlcpugw.denonia[.]xyz

IP Addresses

116.203.4[.]0
162.55.241[.]99
148.251.77[.]55