ls (el-ess not i-ess)

Exploring the QNX shadowed password hash formats

28-12-2015

QNX is a Unix-like real-time operating system originally developed in the 1980s, but was acquired by BlackBerry in 2010. You don't often encounter systems running QNX on corporate networks, but you will find it on many embedded systems. There's very little information on performing security assessments or penetration tests of QNX hosts outside of a few brief blog posts [1].

UPDATE: Zach Lanier and Ben Nell covered a lot of the QNX internals in their presentations at SecTor 2011 (https://prezi.com/k4vlmnvhibss/a-replicant-by-any-other-name-sector-2011/) and CanSecWest 2014 (https://speakerdeck.com/duosec/no-apology-required-deconstructing-bb10).

If you're fortunate enough to compromise a QNX host during a penetration test or while performing research, you might notice that the `/etc/shadow` shadowed password file contains hashes in an unusual format. This post should help you understand what you're looking at and understand how to crack these hashes.

Printable hash format

The `/usr/bin/passwd` binary on a QNX system generates the printable password hash stored in `/etc/shadow`. It contains all the logic needed to generate and parse these hashes, and is the best source of information for reverse engineering the format.

UPDATE: Solar Designer (@solardiz (https://twitter.com/solardiz)) pointed out that there's also DES support. Not sure how I missed that when I first wrote this. Oops! I'll go back over my notes in the coming days and update this post with the missing information.

There are several hash functions supported: MD5, SHA-{256,512} and the insecure (CVE-2000-0250 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0250)) legacy QNX crypt implementation. These hash functions are called using the functions `md5_crypt`, `sha2_crypt` and `qnx_crypt` respectively.

There is also support for parsing plain-text passwords using the `plain_crypt` function, but it seems you must create these `/etc/shadow` entries manually. Interestingly, there is SHA-1 support in a collection of functions prefixed with `sha1_`, but there is no call path in the binary for them and they cannot be used.

The default hash function in QNX Neutrino 6.6.0 is SHA-512 with 1000 rounds and a salt of 16 bytes [2].

Generating a SHA-512 hash for a user account with the password `password` with 1000 rounds and a salt of 8 bytes results in:

```
username:@S@386d...truncated...da5d@129b6761:1448613322:0:0
```

The format shown above is common for MD5 and SHA-{256,512}. If you choose the QNX crypt hash function, the output is similar to a traditional Linux DES crypt string.

Entries in the shadowed password file are first delimetered by `:` and split into the following sections:

- `username` -- user account name
- `@S,100@386d...truncated...da5d@129b6761` -- printable hash string (varies depending on the hash function used)
- `1448613322` -- Unix epoch timestamp of when the password was set
- `0` -- unknown
- `0` -- unknown

I haven't checked what the last two values are used for. They always appear to be zero and may be related to disabled accounts.

The actual printable hash string also delimetered by `@` and split into the following sections:

- `S,100` -- hash function used followed by the number of rounds (comma-separated)
    - `S` -- SHA-512
    - `s` -- SHA-256
    - `m` -- MD5
    - `p` -- plain-text password
- `386d...truncated...da5d` -- hash function output as a hexadecimal string
- `129b6761` -- salt as a hexadecimal string

Password hash examples

All of the below examples are hashes of the password `password`.

SHA-512 with 1000 rounds and a salt of 16 bytes:

```
username:@S@60653c9f515eb8480486450c82eaad67f894e2f4828b6340fa28f47b7c84cc2b8bc
451e37396150a1ab282179c6fe4ca777a7c1a17511b5d83f0ce23ca28da5d@caa3cc118d2deb23:
1448585812:0:0
```

SHA-512 with 1000 rounds and a salt of 8 bytes:

```
username:@S@386d4be6fe9625c014b2486d8617ccfc521566be190d8a982b93698b99e0e3e3a18
464281a514d5dda3ec5581389086f42b5dde023e934221bbe2e0106674cf7@129b6761:14485858
64:0:0
```

SHA-256 with 1000 rounds and a salt of 16 bytes:

```
username:@s@1de2b7922fa592a0100a1b2b43ea206427cc044917bf9ad219f17c5db0af0452@36
bdb8080d25f44f:1448585954:0:0
```

MD5 with 1000 rounds and a salt of 16 bytes:

```
username:@m@bde10f1a1119328c64594c52df3165cf@6e1f9a390d50a85c:1448585838:0:
```

Inside the binary

The `/usr/bin/passwd` binary was taken from a QNX Neutrino 6.6.0 system and isn't very exciting. It's a 32-bit ELF executable, not stripped and dynamically linked against `libc`, `ld-linux` and `linux-gate`. It implements all the hash functions used in the shadowed password file.

The file `/etc/default/passwd` influences the behaviour of the binary. QNX crypt functions are only used if the `QNXCRYPT` directive is found in this file. The other interesting directives are `STRICTPASSWORD` that ensures passwords use at least two character sets and `NOPASSWORDOK` that allows blank passwords.

A random salt is produced by the function `gensalt` and uses the current system time to initialise a custom random algorithm. It uses `initstate`, `setstate`, `random` and `srandom` functions for those that want to dig deeper. I didn't look at this in great detail, but it fails if it gathers less than 8 bytes of entropy for the random operations. The salt is a hexadecimal string; 16 characters if set for 16 bytes.

`qnx_crypt` has been documented before and appears unchanged in this binary [3].

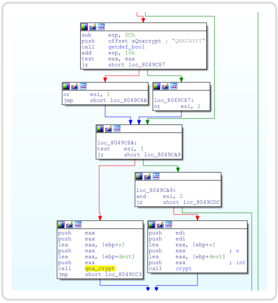
`md5_crypt` uses the `MD5Init`, `MD5Update`, `MD5Transform` and `MD5Update` functions.

`sha2_crypt` uses the `shaXXX_init`, `shaXXX_update` and `shaXXX_done` functions where `XXX` is the bit size e.g. 512.

The QNX concept of hash function rounds is "off-by-one". The hash functions are initialised and then updated as follows:

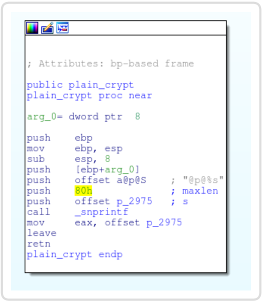
```
digest = update(salt), update(password) * rounds, update(password)
```

The hash function is updated with the password one last time in case the number of rounds is set to zero, so for 1000 rounds it's actually 1001 rounds before the digest is calculated. Other than that, it's all pretty standard.



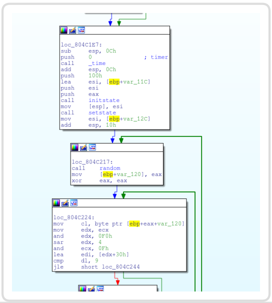
```
sub     esp, 0Ch
push    offset aQnxcrypt ; "QNXCRYPT"
call    getdef_bool
add     esp, 10h
test    eax, eax
jz      short loc_8049C87

or      esi, 1
jmp     short loc_8049C8A

loc_8049C87:
or      esi, 2

loc_8049C8A:
test    esi, 1
jz      short loc_8049CA9

loc_8049CA9:
and     esi, 2
jz      short loc_8049CDC

push    eax
push    eax
lea     eax, [ebp+s]
push    eax
lea     eax, [ebp+dest]
push    eax
call    qnx_crypt
jmp     short loc_8049CC3

push    edi
push    edi
lea     eax, [ebp+s]
push    eax                     ; s
lea     eax, [ebp+dest]
push    eax                     ; int
call    crypt
```

(/images/qnx_default_directives.png)



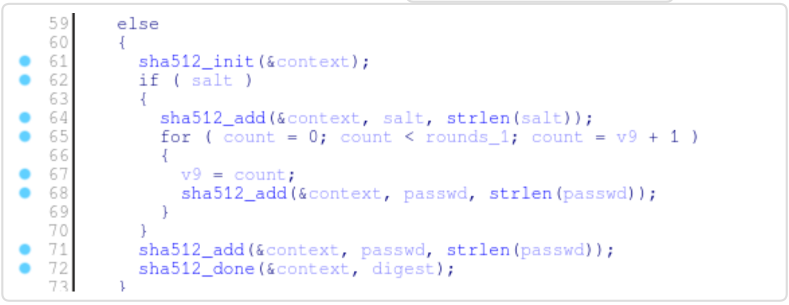
```
; Attributes: bp-based frame

public plain_crypt
plain_crypt proc near

arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
push    [ebp+arg_0]
push    offset a@p@S     ; "@p@%s"
push    80h              ; maxlen
push    offset p_2975    ; s
call    _snprintf
mov     eax, offset p_2975
leave
retn
plain_crypt endp
```

(/images/qnx_plain_crypt.png)



```
loc_804C1E7:
sub     esp, 0Ch
push    0                ; timer
call    _time
add     esp, 0Ch
push    100h
lea     esi, [ebp+var_11C]
push    esi
push    eax
call    initstate
mov     [esp], esi
call    setstate
mov     esi, [ebp+var_12C]
add     esp, 10h

loc_804C217:
call    random
mov     [ebp+var_120], eax
xor     eax, eax

loc_804C224:
mov     cl, byte ptr [ebp+eax+var_120]
mov     edx, ecx
and     edx, 0F0h
sar     edx, 4
and     ecx, 0Fh
lea     edi, [edx+30h]
cmp     dl, 9
jle     short loc_804C244
```

(/images/qnx_random.png)



```
else
{
  sha512_init(&context);
  if ( salt )
  {
    sha512_add(&context, salt, strlen(salt));
    for ( count = 0; count < rounds_1; count = v9 + 1 )
    {
      v9 = count;
      sha512_add(&context, passwd, strlen(passwd));
    }
  }
  sha512_add(&context, passwd, strlen(passwd));
  sha512_done(&context, digest);
}
```

(/images/qnx_sha512_crypt.png)

Cracking QNX password hashes

John the Ripper (even with the jumbo patches) and other common tools do not support the QNX shadowed password hash format.

I tried to reimplement the logic of the hash functions used by QNX using the Python modules `hashlib` and `passlib`, as well as playing around with some other non-Python options. I couldn't ever get my output to match what the `/usr/bin/passwd` binary produced. All my approaches were consistent, however, so I assume there is something unique about the QNX hash function implementations; they are custom and do not rely on common external libraries, such as OpenSSL. (I could also be consistently incorrect in my approach.)

I'm not a crypto guy nor did I spend much time on reversing the hash functions I saw, but I did look at the SHA-2 reference documentation and the QNX implementation appeared to be doing the right thing. It feels like the output differences are an oddity or an intentional tweak in their proprietary implementation.

I need to update this section when I'm less lazy and feel like rewriting my notes.

TL;DR If you want to brute force these hashes, you can script up a GDB session to call functions from the binary itself (see below). A more elegant solution is to use `dlopen(3)` and `dlsym(3)` to call the functions from a C wrapper or Python using `ctypes` or `CFFI`.

```
Breakpoint 1, 0x080493c1 in main ()
(gdb) call sha2_crypt(512, "password", "abcd1234abcd1234", "1000")
$1 = 134559360
(gdb) x/s $1
0x8053680 <out.3087>:   "@S@1030f372de34b8caac99b481d81ad9b57b923b385edcd3ed84f
6721192f5238f34aba739e1d124919bd85c8efe13948593a6b691d8b41c1be5bc9b3906577f5d@a
bcd1234abcd1234"
```

References

[1] The Pentesting QNX Neutrino RTOS (https://www.fishnetsecurity.com/6labs/blog/pentesting-qnx-neutrino-rtos) blog post from FishNet Security was the best information I could find on QNX for security assessments or penetration tests.

[2] The defaults can be found from the `/usr/bin/passwd` binary itself or in the online QNX development references (http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.utilities%2Ftopic%2Fp%2Fpasswd.html).

[3] The qnx_decrypt.c code updated by SilentDream (https://github.com/SilentDream/scripts/blob/master/qnx_decrypt.c) is the best reference for understanding QNX crypt hashes.