Primitive Math

Dcoder

1 Introduction

The “Primitive Math” challenge by promix17 [2] requires a number of different skills to solve. It offers anti-debugging, function-level code encryption, a virtual machine and some, well, primitive math.

The first thing this challenge does is unpack itself into 2 other binaries: `Primitive_Math.exe` and `svchost.exe`. In what looks like a bad attempt at malware, the `svchost.exe` binary, stored in the local temporary file directory, looks around for running OllyDBG and IDA processes, and tries to tamper with them in case it does. This is easy to work around, and I will make no further mention of it. The `Primitive_Math.exe` process is where the actual interesting code is, and where we are going to focus.

Most of the important functions are encrypted using an elementary `xor` cipher. They are decrypted at runtime, using a preamble similar to the following:

```
loc_401819:
                mov     esi, offset byte_430724
                mov     edi, offset loc_401839
                mov     ecx, offset loc_4018A2
                sub     ecx, edi

loc_40182A:
                db      3Eh
                mov     dl, [edi]
                db      3Eh
                mov     al, [esi]
                xor     dl, al
                db      3Eh
                mov     [edi], dl
                inc     edi
                inc     esi
                loop    loc_40182A
```

One way to let the program decrypt itself completely is to take the re-encryption step and re-purpose it to fill the encryption loop with `0x90`. This way, the function is decrypted once, and remains subsequently unhidden.

2 First blood

The first checks we are able to spot happen in the message handling function for the main window, when the “Enter” button is pressed (starting at `40284D`). In here, it is verified

Character	Function
X	Push input
A	Add
M	Subtract
P	Multiply
D	Divide
S	Sine
C	Cosine
G	Hyperbolic Sine
H	Hyperbolic Cosine
W	Exponentiate
0–9	Push digit

Table 1: Serial characters and associated VM functions.

that the user name has at least 6 characters and the serial has 13. Further down, we find some additional checks:

1. The first character of the serial must equal 'A' $+ \sum_i u_i \bmod 20$, where u_i is the ith character of the username.
2. The second character of the serial must equal 'A' $+ \sum_{i=2}^{13} s_i \bmod 20$, where s_i is the ith digit of the serial.
3. The 12th character of the serial must equal '0' $+ \sum_i (u_i \oplus i) \bmod 10$[1].
4. Let $|X|$ be the number of distinct 'X' characters in the serial. Then, $6 \cdot |X| \equiv 1 \pmod{11}$.
5. Let $|P|$ and $|A|$ be the number of distinct 'P' and 'A' characters in the serial, respectively. Then, $(|X| + |P| + |A|)^2 \bmod 256 = 16$.

The 5th check's solution, 16, was found by trial and error: this result is used to decrypt the next serial check. Since there are only so many possible squares and values to test, finding it was quite straightforward.

3 Push a Push Pop

Once we get the basic checks out of the way, we get to the main course, starting at address `4018B0`. Here we find a nice twist: the serial's characters are used as instructions in a basic stack-based floating-point VM, not unlike x87!

This VM is quite simple, and only performs basic arithmetic functions. Table 1 lists the serial characters and respective instructions executed in the VM.

The program now uses the VM, which executes a function $f(x)$ directly from the serial, to verify the following identity:

[1] $\oplus$ means `xor`.

$$\frac{f(i+\epsilon)-f(i-\epsilon)}{2\epsilon} = g(x), \qquad i \in \{0,1,\ldots,9\},$$

where $\epsilon = 1 \times 10^{-7}$ and

$$g(x) = \left(\cos x\sqrt{x} + \frac{\frac{1}{2}\sin x}{\sqrt{x}}\right)\cosh(\sin x\sqrt{x}).$$

If you know a thing or two about Calculus (if not, cf. [1, 3]), you'll immediately notice that $g(x)$ must be the derivative of $f(x)$ for the identity to hold (for arbitrarily small ϵ, mind you). Thus, $f(x)$ can be given by the anti-derivative (or indefinite integral) of $g(x)$, namely

$$f(x) = \int g(x)dx = \sinh(\sqrt{x}\sin x) + C, \qquad C \in \mathbb{R}.$$

This function can be implemented in the challenge's VM quite easily, with the sequence `12DXWXSPGCA`. We do not control `C`, for it is dependent on the user name. But it nicely maps to the addition of an arbitrary constant C to the anti-derivative, as shown in the above equation. Notice, also, how rules 4 and 5 of Section 2 are respected, since $6 \times 2 \bmod 11 = 1$ and $(2+1+1)^2 \bmod 256 = 16$.

Finally, the program checks the identity

$$\sum_{i=0}^{100}\left(\left\lfloor 10(f(\frac{i}{10}) - C)\right\rfloor \oplus i\right) \bmod 256 = 39,$$

thus ensuring that the correct function $f(x)$ has been used.

References

[1] Hardy, G. H.: *A Course of Pure Mathematics*. Cambridge University Press, 10th edition, 1967, ISBN 0521092272.

[2] promix17: *Primitive Math*. `http://crackmes.de/users/promix17/primitive_math/`, March 2011.

[3] Spivak, Michael: *Calculus*. Publish or Perish, 3rd edition, 1994, ISBN 0914098896.

A Full solution code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int crc1(char *str, int len)
{
  int i,h;
  for(i=0,h=0; i < len; ++i)
    h += str[i];
  return h%20+'A';
}

int crc2(char *str, int len)
{
  int i,h;
  for(i=0,h=0; i < len; ++i)
    h += str[i] ^ i;
  return h%10;
}

int main(int argc, char **argv)
{
  char serial[16] = {0};
  int i, h ;

  if(argc != 2)
  {
    printf("Usage: %s <username>\n", argv[0]);
    return -1;
  }

  sprintf(serial, "12DXWXSPG%dA", crc2(argv[1], strlen(argv[1])));
  printf("%c%c%s\n", crc1(argv[1], strlen(argv[1])), crc1(serial, strlen(serial)), serial);

  return 0;
}
```