Introduction to Hacking Thick Clients: Part 6 – The Memory

written by Austin Altmann | June 18, 2020

Introduction to Hacking Thick Clients is a series of blog posts that will outline many of the tools and methodologies used when performing thick client security assessments. In conjunction with these posts, NetSPI has released two vulnerable thick clients: BetaFast, a premier Betamax movie rental service, and Beta Bank, a premier finance application for the elite. Many examples in this series will be taken directly from these applications, which can be downloaded from the BetaFast GitHub repo. A brief overview is covered in a previous blog post.

Installments:

1. The GUI
2. The Network
3. The File System and Registry
4. The Assemblies
5. The API
6. The Memory

Overview

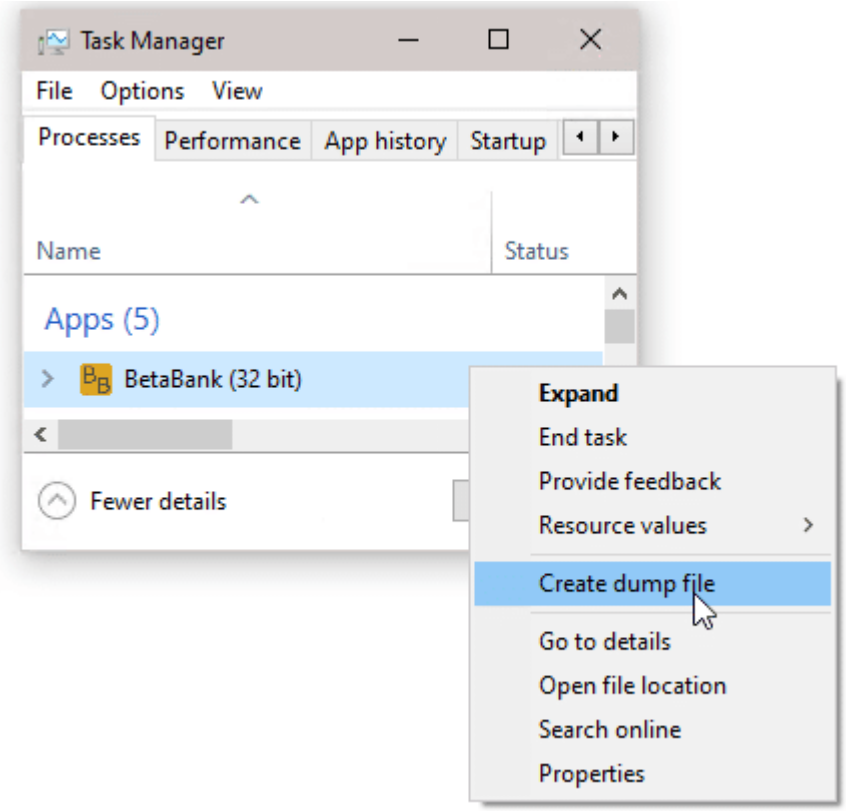
Welcome to the *final* installment of Introduction to Hacking Thick Clients! What a journey. We've learned a lot, but there's just one last thing to mention – the memory!

Viewing Data

If a system is compromised by an attacker, values stored in memory may be exposed. Of course, if an attacker has compromised a system to the point of analyzing memory, there are *a lot* of other issues. But it is my opinion that an application should take responsibility for its security as

much as possible and not rely on the security of the system it resides on. I go further into some solutions and their practicalities in my freshman blog post, but most of them are focused on *limiting* exposure, not *eliminating* exposure. If possible, assign sensitive data to mutable data types, such as arrays. Immutable data types like strings cannot be cleared from memory on command, but an array can be cleared from memory as soon as its values are no longer needed.

So let's see how Beta Bank handles its sensitive data. I'll let it slide if I've logged in and the password is still in memory. But in this example, I've already authenticated to the application, logged out, and meandered through my studio apartment for a few minutes. That's enough leeway – time to examine the application's memory. First, find the application in Task Manager, right click, and click "Create dump file."

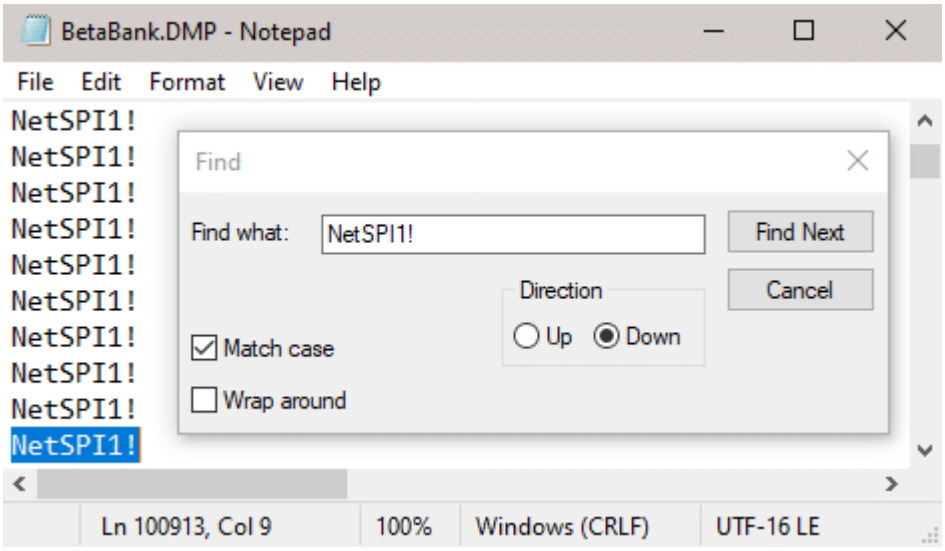


Now, a likely massive file has been generated. That's everything from that process's memory. And thankfully, friendly neighborhood Mark Russinovich has created a tool called Strings that will help to analyze this data by

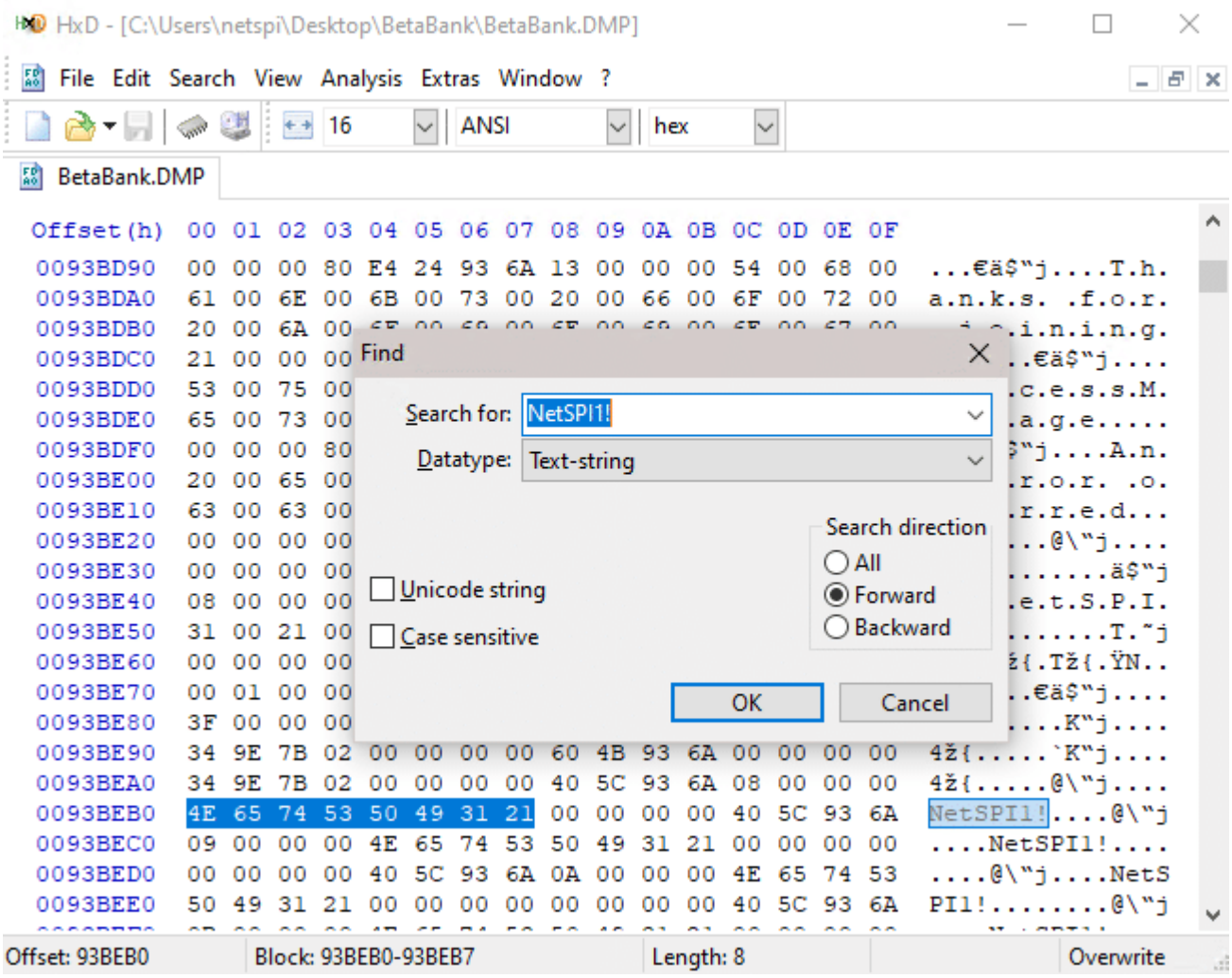
retrieving only the string values. I've taken the liberty of running the following line in PowerShell:

```
.\strings.exe .\BetaBank.DMP | Out-File -FilePath .\BetaBank.DMP.txt
```

The resultant file should be a much more manageable size, and any text editor worth its salt can search for values of choice. The password I used when logging in was NetSPI1!. And it shows up a lot.



Another choice is to just open the entire dump file in a hex editor like HxD. The password appears just as easily and visibly, but so does a lot of other data.

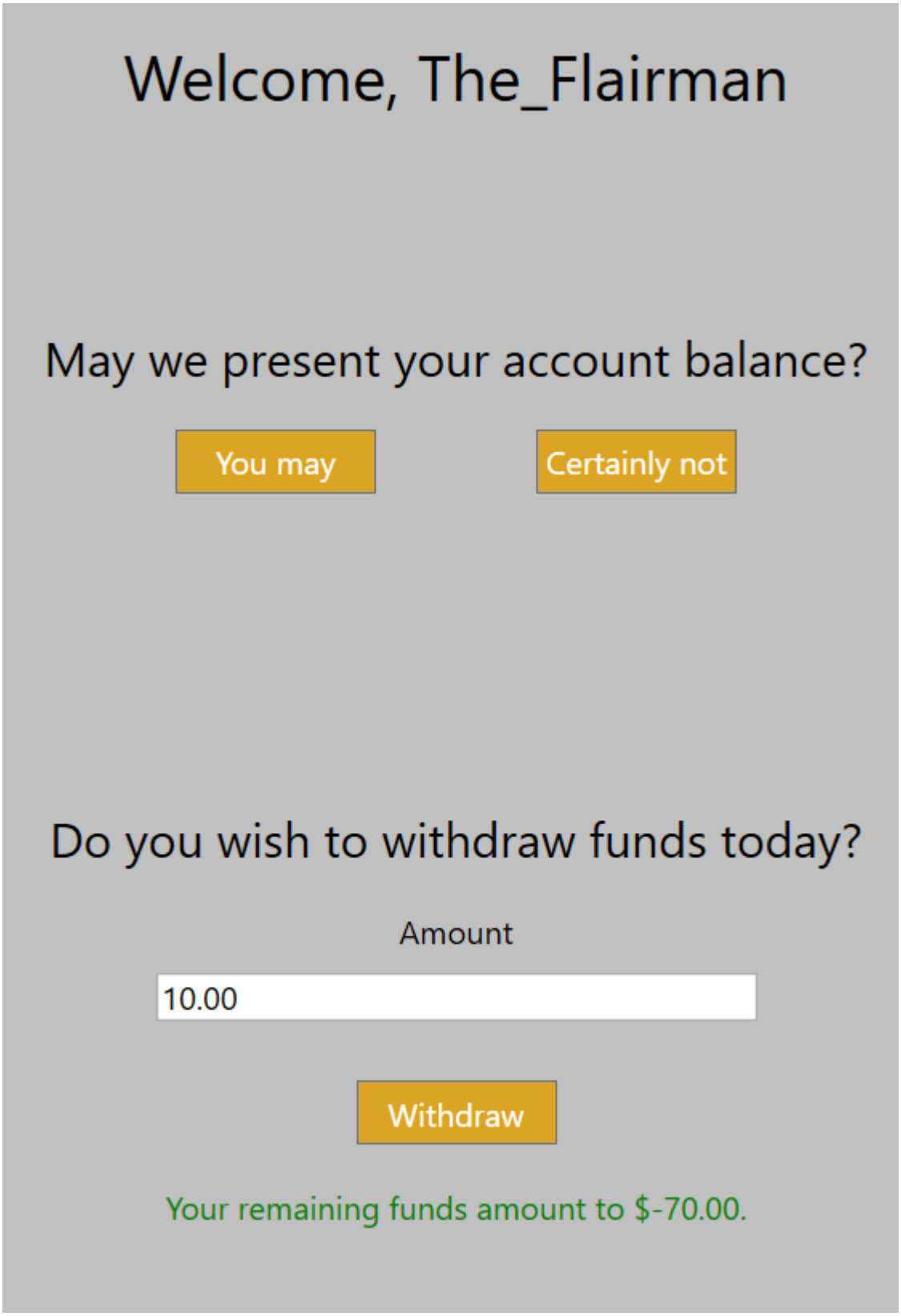


Modifying Data

As I was coming up with examples for this portion of the blog, I realized that I first began modifying data in memory as a delinquent child. If you were at all like me growing up and wanted to impress people with lies, you may remember Cheat Engine. Just like my Beta Bank password, game data such as item counts, experience points, and high scores may be in memory plain as day. And these values could be located through a simple search, and the data at that address could be changed to a new value. That level 25 character was a few clicks away from being level 2,147,483,647.

These days, it is much more difficult to earn the admiration of my peers through lies. But gosh darn it, I'm going to try. I've heard from previous installments in this blog series that there's a user named The_Chairman who has a lot of money. I'm

going to withdraw a lot of it. The plan is to change my username in memory to The_Chairman. Since the space for the username will already be allocated, I've created an account with the same username length – The_Flairman.

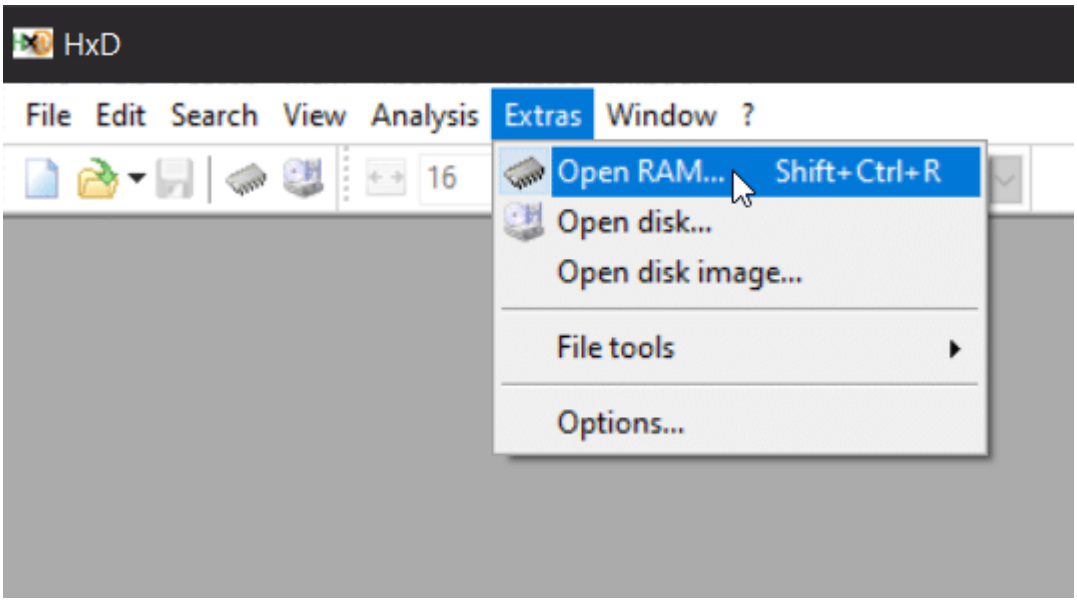


This account is going to be swamped with overdraft fees. In

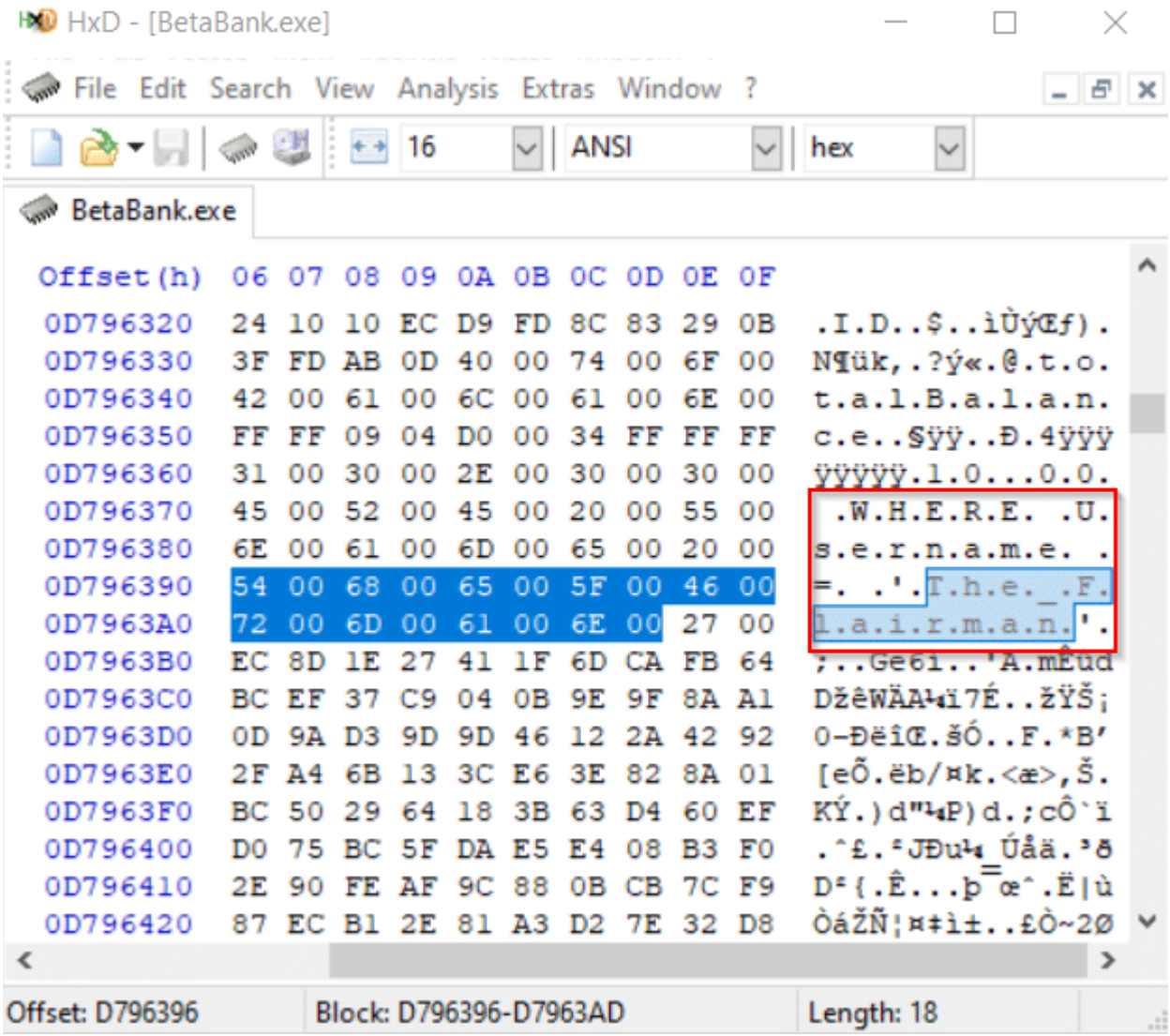
Wireshark, a SQL statement to withdraw money is being sent with the username The_Flairman. For the sake of this example, let's just use this as verification and pretend the traffic is encrypted. And that a bunch of other security vulnerabilities are patched while still allowing me to do something like this.

```
0000  00 11 22 33 44 55 00 05  9a 3c 7a 00 08 00 45 00   ..."3DU.. .<z...E.
0010  00 f2 d0 dd 40 00 80 06  10 dd 0a 80 01 93 0a 02   ....@... ........
0020  02 37 5a 3c 05 99 ef 20  a1 31 fe 95 96 6d 50 18   .7Z<... .1...mP.
0030  01 fe f1 9f 00 00 01 09  00 ca 00 00 01 00 16 00   ........ ........
0040  00 00 12 00 00 00 02 00  00 00 00 00 00 00 00 00   ........ ........
0050  01 00 00 00 55 00 50 00  44 00 41 00 54 00 45 00   ....U.P. D.A.T.E.
0060  20 00 42 00 65 00 74 00  61 00 42 00 61 00 6e 00    .B.e.t. a.B.a.n.
0070  6b 00 42 00 61 00 6c 00  61 00 6e 00 63 00 65 00   k.B.a.l. a.n.c.e.
0080  73 00 20 00 53 00 45 00  54 00 20 00 42 00 61 00   s. .S.E. T. .B.a.
0090  6c 00 61 00 6e 00 63 00  65 00 20 00 3d 00 20 00   l.a.n.c. e. .=. .
00a0  42 00 61 00 6c 00 61 00  6e 00 63 00 65 00 20 00   B.a.l.a. n.c.e. .
00b0  2d 00 20 00 31 00 30 00  2e 00 30 00 30 00 20 00   -. .1.0. .0.0. .
00c0  57 00 48 00 45 00 52 00  45 00 20 00 55 00 73 00   W.H.E.R. E. .U.s.
00d0  65 00 72 00 6e 00 61 00  6d 00 65 00 20 00 3d 00   e.r.n.a. m.e. .=.
00e0  20 00 27 00 54 00 68 00  65 00 5f 00 46 00 6c 00    .'.T.h. e._.F.l.
00f0  61 00 69 00 72 00 6d 00  61 00 6e 00 27 00 3b 00   a.i.r.m. a.n.';.
```

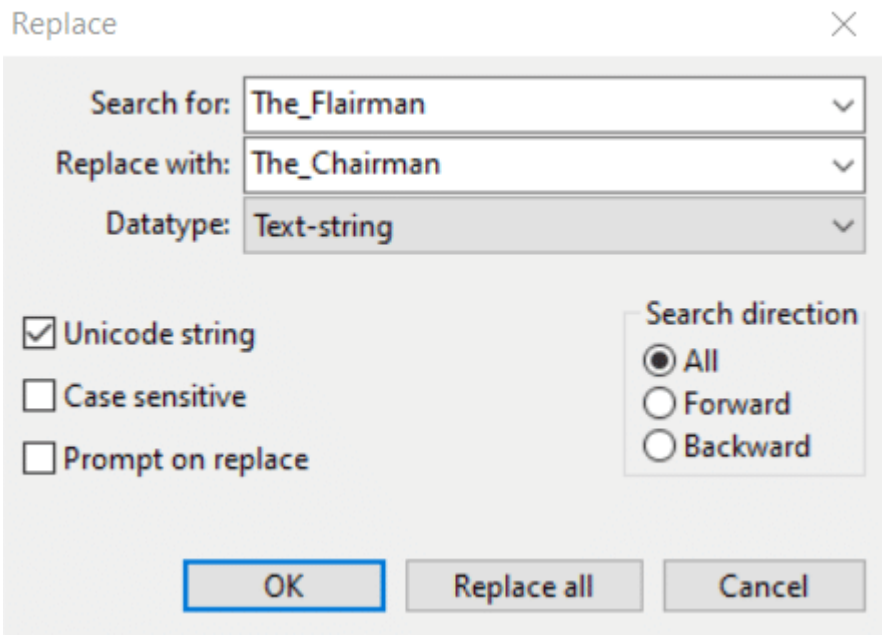
HxD is just as usable for editing memory as it is viewing, so we'll stick with what we know. With Beta Bank running, click Open RAM to view the live memory for the process.



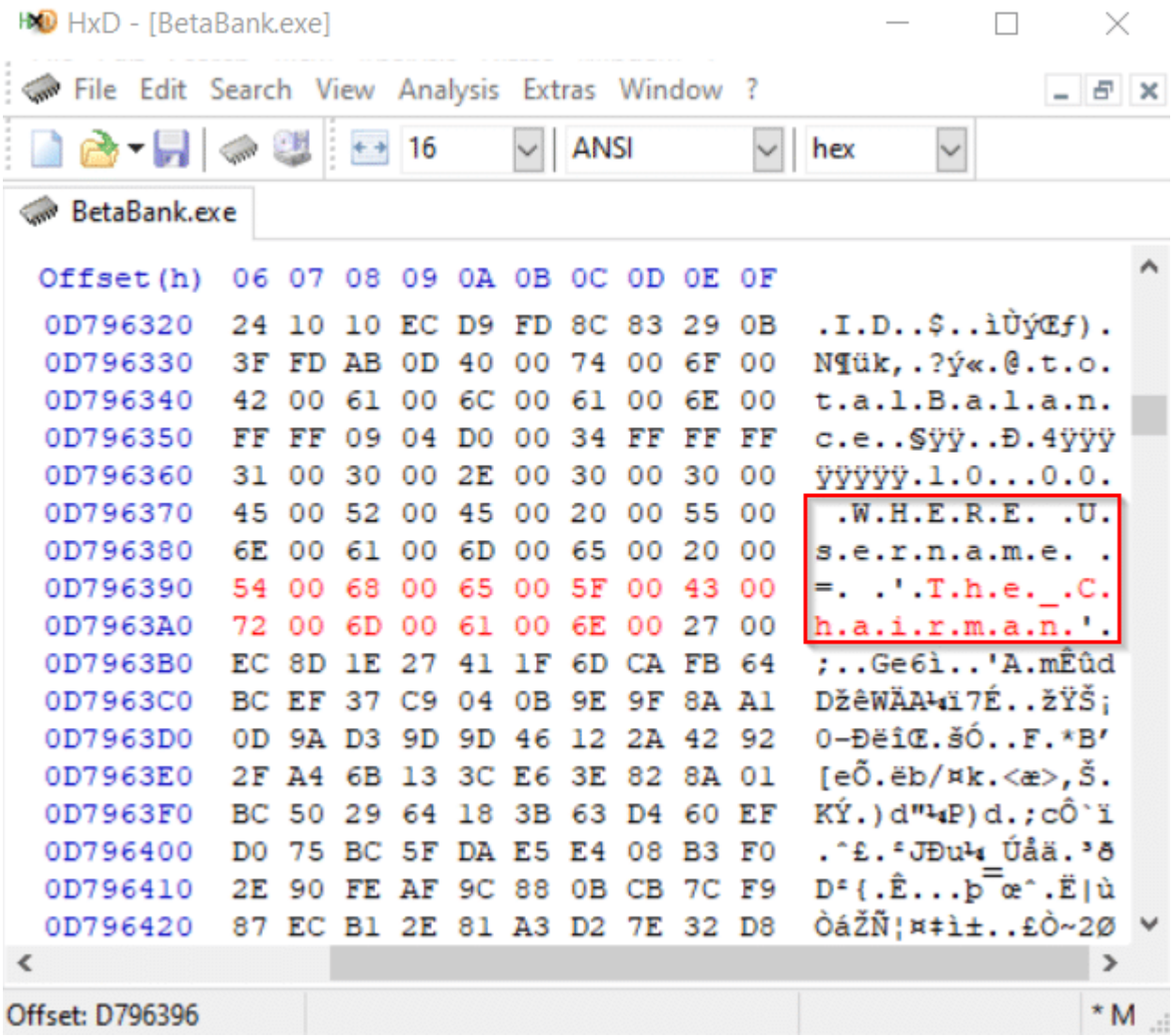
Searching for The_Flairman shows a number of locations where the username is referenced in memory.



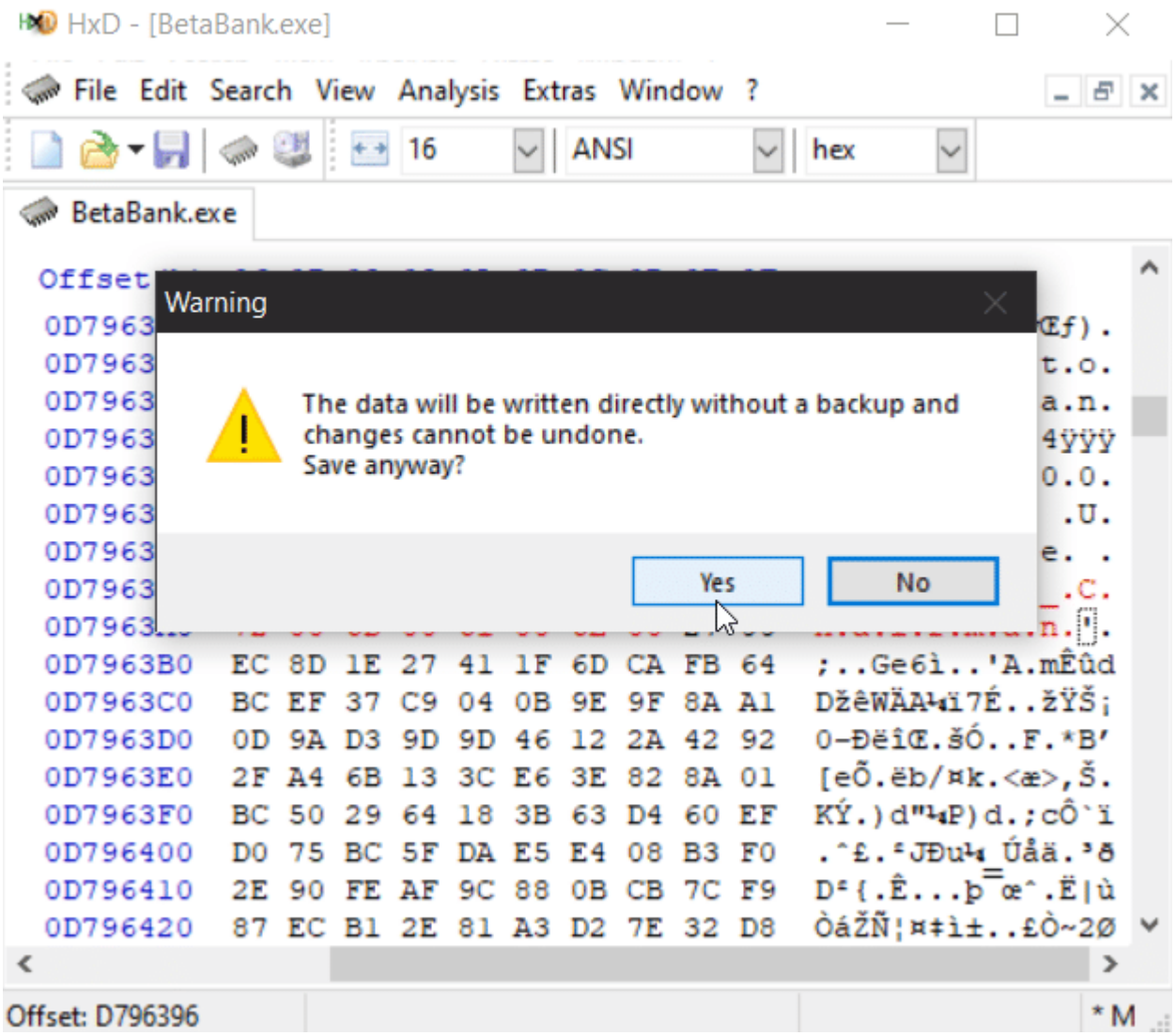
I went ahead and replaced every instance of The_Flairman with The_Chairman.



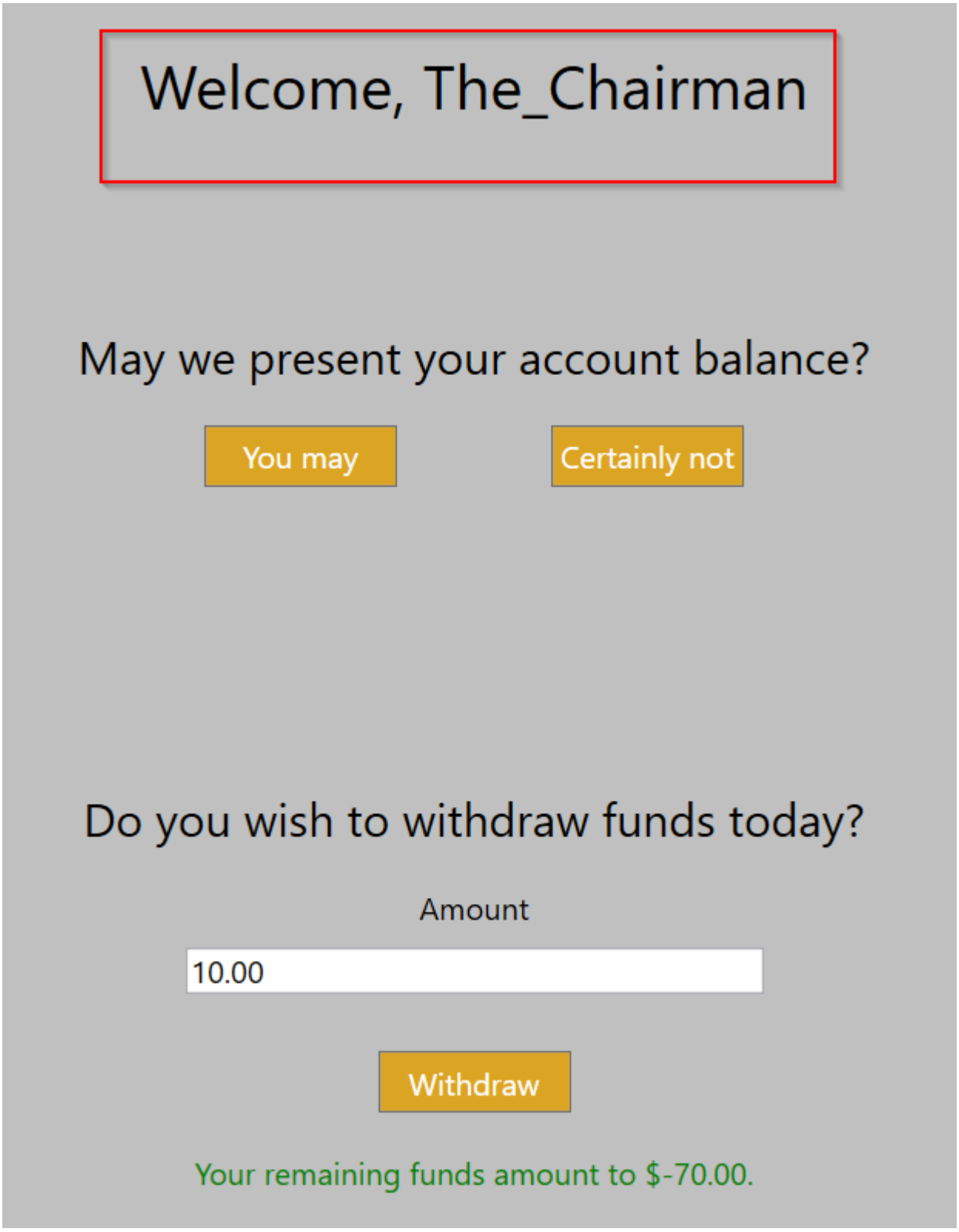
Below is one of these instances.



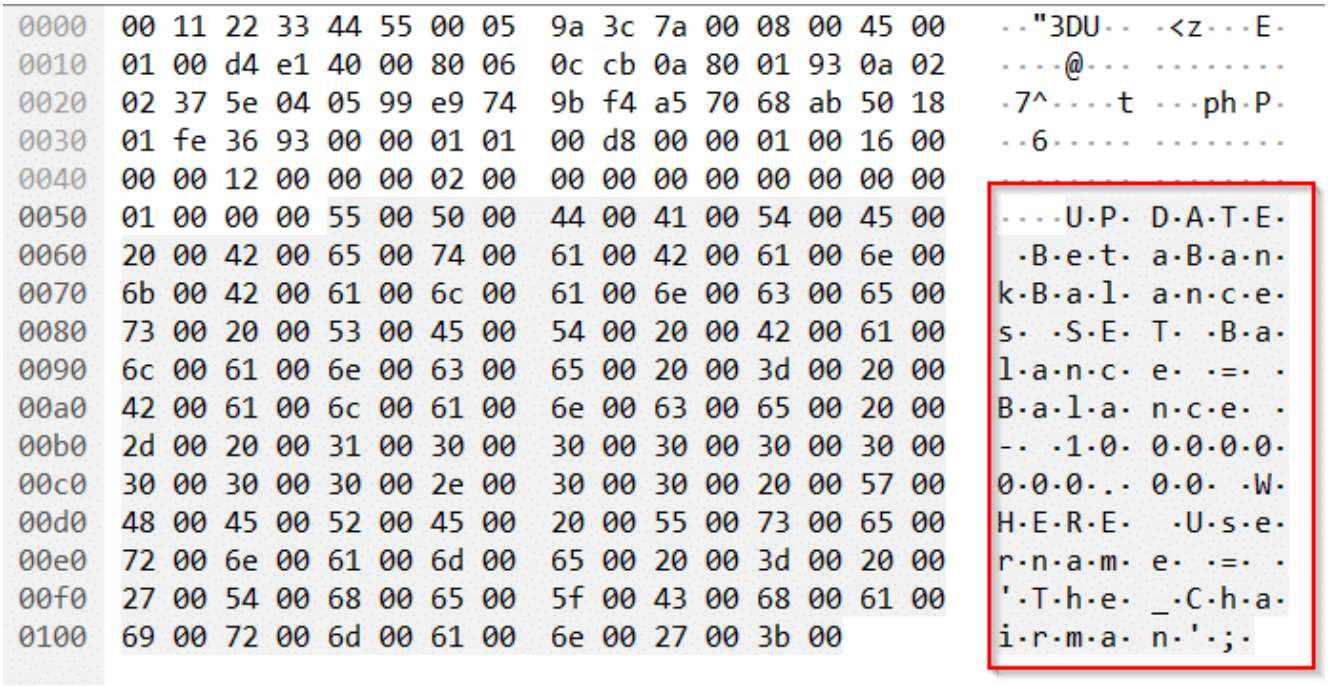
Saving these edits will overwrite the current live memory.



Back in the application, it’s clear that these changes went into effect. But were they functional?



Using Wireshark, I was able to verify that the Withdraw button will now send a SQL command to withdraw a large amount of money with the username The_Chairman.



This wouldn't be an issue at all if the SQL command wasn't written with a complete lack of security awareness. In this application, there's not a lot going on that would be "vulnerable" to memory values changing since authorization and storage is all on the server. Others may have simple boolean controls that can be bypassed through memory editing. Plenty of client-side control logic could be bypassed. I even found a simple example involving WinRAR, everyone's favorite application, and how to bypass its registration control.

Further Reading

This blog post barely scratched the surface of memory analysis. The techniques outlined above are a few that are used during our routine testing of thick client applications. But they're also entry points for diving further into malware analysis and reverse engineering. The following resources may be helpful starting points for additional learning:

- PowerShellArnsenal
- Volatility Framework
- A debugger such as x64dbg

Introduction to Hacking Thick Clients: The Conclusion

This series outlined quite a few thick client application testing methods and tools at a high level. While it never went especially deep on any one topic, I hope that it, along with BetaFast and Beta Bank, have opened the door for people to develop new security testing skills while having a bit of fun. This is a service line that really pushes creative thinking. Some of the exploitation steps I've seen in reports have made me reach out to coworkers and ask how they even thought to try them. Every once in a while, they unmute me and reply. But I digress . . .