Research Threat intelligence Microsoft Defender Vulnerabilities and exploits

18 min read

Patching Perforce perforations: Critical RCE vulnerability discovered in Perforce Helix Core Server

By Microsoft Threat Intelligence

December 15, 2023

Microsoft Defender External Attack Surface Management Remote code execution

Microsoft discovered, responsibly disclosed, and helped remediate four vulnerabilities that could be remotely exploited by unauthenticated attackers in Perforce Helix Core Server ("Helix Core Server"), a source code management platform largely used in the videogame industry and by multiple organizations spanning government, military, technology, retail, and more. Helix Core Server customers are strongly urged to update to version 2023.1/2513900 or upgrade to the 2023.2 version, available here: https://www.perforce.com/downloads/helix-core-p4d. The most critical of the four vulnerabilities has a CVSS score of 10.0 because it allows for arbitrary remote code execution as *LocalSystem* by unauthenticated remote attackers. An attacker with system-level remote code execution access to a source code management platform can insert backdoors into software products, exfiltrate source code and other intellectual property, and pivot to other sensitive enterprise infrastructure. While Microsoft has not observed evidence of in-the-wild exploitation for any of these vulnerabilities, exploitation of the most critical vulnerability could give unauthenticated attackers complete control over unpatched systems and connected infrastructure.

Due to the way Microsoft's deployed Helix Core Server were configured, at no point were any of Microsoft's internet-facing servers vulnerable to this critical vulnerability. No consumer, customer, or partner data was at risk or leaked.

Microsoft's commitment to gaming and community security is paramount, and we worked closely with Perforce to report these vulnerabilities and drive remediation. We thank Perforce and are grateful for their team's quick response in developing and releasing patches for these vulnerabilities.

While the three high severity vulnerabilities could be used to launch attacks such as a denial of service (DoS) against vulnerable systems, vulnerabilities with a CVSS score of 10.0 have the most severe potential impact that can extend beyond the vulnerable component, introducing a risk to software supply chains. The discovered vulnerabilities are summarized in the table below:

CVE ID	CVSS Score	CWE ID	Vulnerability
CVE-2023-5759	7.5	CWE-405: Asymmetric Resource Consumption (Amplification)	Unauthenticated DoS via RPC Header Abuse
CVE-2023-45849	10.0	CWE-306: Missing Authentication for Critical Function	Unauthenticated Remote Code Execution as LocalSystem via user-bgtask RPC Command
CVE-2023-35767	7.5	CWE-306: Missing Authentication for Critical Function	Unauthenticated DoS via rmt-Shutdown RPC Command
CVE-2023-45319	7.5	CWE-252: Unchecked Return Value	Unauthenticated DoS via rmt-UpdtFovrCommit RPC Command

Helix Core Server listens on TCP port 1666 by default, though server administrators will often change this port number to hide from scanners or to host Helix Core Server via TLS. Microsoft scanned the internet in November 2023 for TCP port 1666 with a custom Helix Core Server network signature and found over 1,000 exposed Perforce Helix Core Server instances.

In this blog, we detail how we discovered each of the vulnerabilities and highlight the potential impact if exploited. Alongside applying Perforce's patches, we also include additional mitigation and protection guidance for customers to minimize the risk of exploitation. Lastly, we're sharing this information with the broader community to drive awareness to further improve protections across the security ecosystem, and to emphasize the importance of responsible disclosure and collaboration to secure platforms and devices.

Discovering the vulnerabilities

To keep Microsoft's game development studios and their customers safe, we recently conducted an application security review of Helix Core Server, the source code management platform relied on by most of our studios. For our security review, we analyzed Helix Core Server version 2023.1.244.2900 and installed it on Windows 11 22H2. We used Helix Core Server's default installation options, which resulted by-design in the Helix Core Server service running as *LocalSystem*:

```
C:\>sc qc Perforce
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: Perforce
        TYPE               : 10  WIN32_OWN_PROCESS
        START_TYPE         : 2   AUTO_START
        ERROR_CONTROL      : 1   NORMAL
        BINARY_PATH_NAME   : "C:\Program Files\Perforce\Server\p4s.exe"
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : Helix Core Server
        DEPENDENCIES       : LanmanWorkstation
        SERVICE_START_NAME : LocalSystem
```

Figure 1. Helix Core Server runs as LocalSystem

Recovering debug symbols

In 2014, Perforce open-sourced the code for their CLI Perforce Client, and informed users we can download the code from the *bin.tools* subdirectory of any given release. While having any source code is invaluable for application security vulnerability research purposes, this source code is specific to the client, not the server. The latter is only distributed in compiled binary form.

The binaries that are installed by Helix Core Server's installer have their debug symbols stripped (removed from the distributed executable images), which makes it harder to understand the disassembled code during static analysis. To aid our review, we attempted to recover these debug symbols.

Discovering debug symbols

Sometimes applications offer software development kits (SDKs) that can be mined for debug symbol data. In the case of Helix Core Server, Perforce offers a "C/C++ API" package for the Windows (x64) platform that comes in the form of a .zip file containing three directories: *include*, *lib*, and *sample*. The *lib* directory is especially interesting for us, as it contains about 400 MB of .lib files:

Name	Size
libclient.lib	19,089 KB
libp4api.lib	199,223 KB
libp4script.lib	118,921 KB
libp4script_c.lib	22,926 KB
libp4script_cstub.lib	1,026 KB
libp4script_curl.lib	11,860 KB
libp4script_sqlite.lib	5,089 KB
librpc.lib	3,445 KB
libsupp.lib	20,698 KB

Figure 2. SDK's .lib files

Like .exe files, .lib files are COFF files that can contain debug symbols. By using *dumpbin.exe /symbols* to inspect each .lib file, we found that the nine .lib files in the package contain a total of 1,251,756 debug symbol entries.

To understand why this is useful to us, let us consider an approximation of how .exe and .lib files are built:

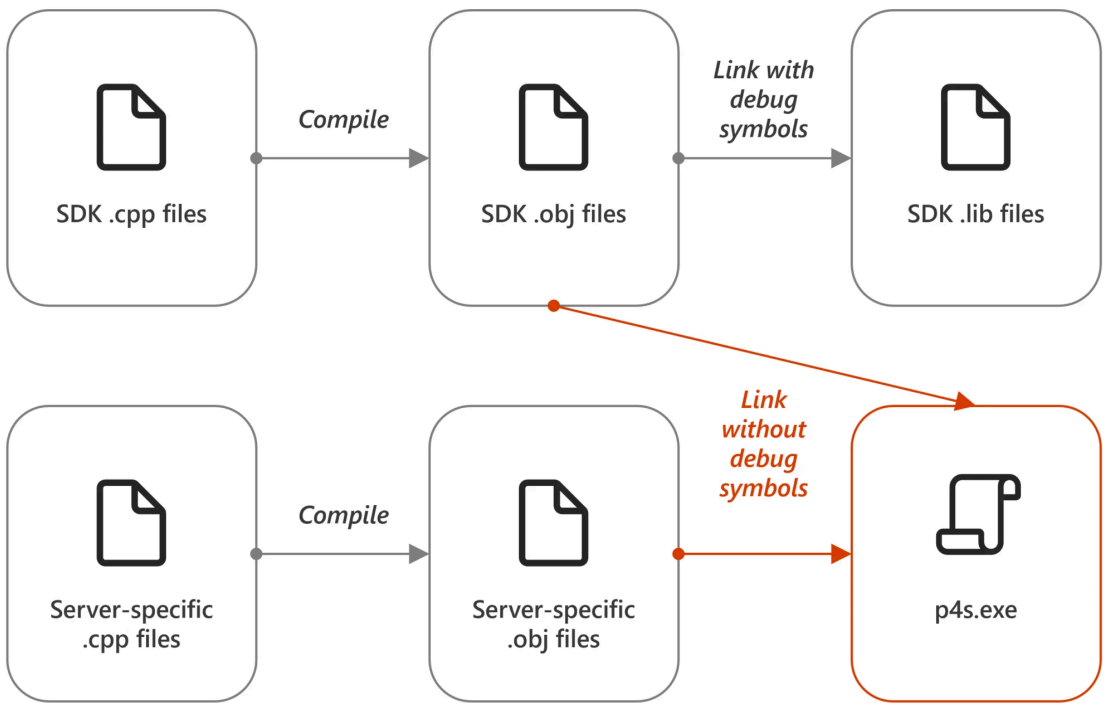


Figure 3. Compilation process

In the diagram above, we can see that the SDK .obj files were linked along with server-specific .obj files to create Helix Core Server's *p4s.exe* ("Perforce Service") file. During that linking process, the debug symbols were stripped. However, the same SDK .obj files had their debug symbols *retained* when linked into the SDK .lib files. Since the .lib files contain debug symbols, we can match each compiled SDK function in each .lib file to its SDK function name. If we can then find those same compiled SDK functions in Helix Core Server's .exe and .dll files, we can map the SDK function names to those functions as well, thus simplifying our analysis of the *p4s.exe* file.

To begin, we must first determine which SDK package to use for our analysis. If we look at the containing directory for the .zip file downloaded from the "C/C++ API" package, we see it contains 144 *p4api* SDK packages:

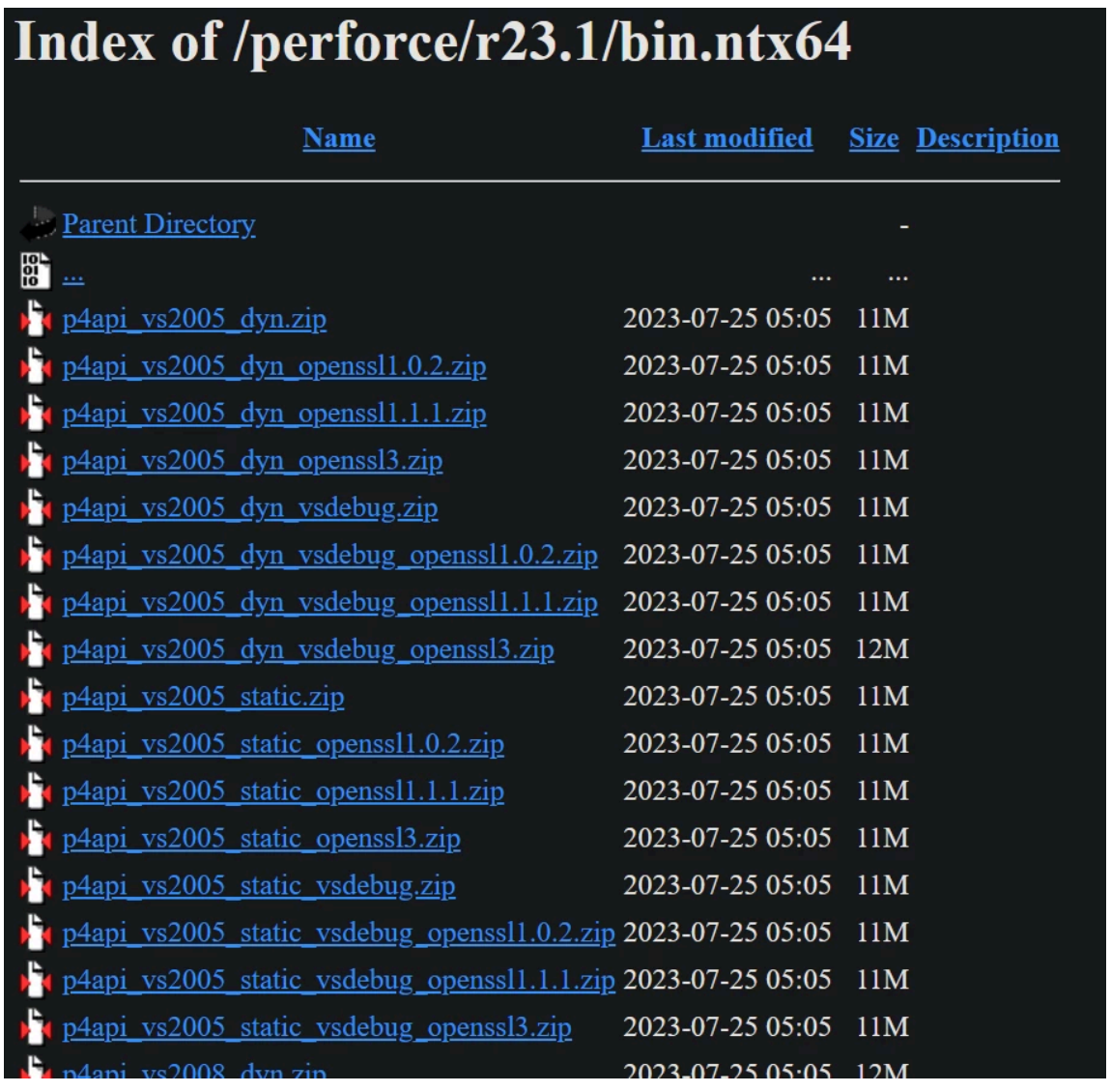


Figure 4. SDK packages

The reason we see 144 packages listed is that there is every combination of the following:

$$\begin{bmatrix} Visual\ Studio\ 2005 \\ Visual\ Studio\ 2008 \\ Visual\ Studio\ 2010 \\ Visual\ Studio\ 2012 \\ Visual\ Studio\ 2013 \\ Visual\ Studio\ 2015 \\ Visual\ Studio\ 2017 \\ Visual\ Studio\ 2019 \\ Visual\ Studio\ 2022 \end{bmatrix} \times \begin{bmatrix} Dynamic \\ Static \end{bmatrix} \times \begin{bmatrix} Release \\ Debug \end{bmatrix} \times \begin{bmatrix} OpenSSL\ 1.0.2 \\ OpenSSL\ 1.1.1 \\ OpenSSL\ 3 \\ None \end{bmatrix}$$

Figure 5. Package combinations

That's nine possible values for compiler, two possible values for linking, two possible values for build type, and four possible values for OpenSSL version. In other words, multiplying those four values together leads us to 144 possible combinations. To map named functions from the SDK's .lib files to Helix Core Server's *p4s.exe* file, we'll need to choose the correct SDK package, since, for example, a function compiled with Visual Studio 2005 may look very different from the same function compiled with Visual Studio 2022.

So how do we know which compiler, linker option, build type, and OpenSSL version were used for our installed distribution of Helix Core Server? We don't. We could make some educated guesses and examine artifacts such as the binaries' Rich Headers to determine the right combination, but instead we chose to use automation to test all possible combinations. (Note that "Rich Headers" is a colloquial term used in the industry, not a Microsoft-official name for this structure.)

Finding the right set of debug symbols

After downloading all of the statically linked *p4api* archives from Perforce's website, we used IDA Pro's F.L.I.R.T. technology to create signatures for each Perforce Helix Core Server SDK package. To do so, we automated the following steps:

1. Use *pcf.exe* ("parsecoff") from IDA Pro's Fast Library Acquisition for Identification and Recognition (FLAIR) SDK to create .pat ("pattern") files for each Perforce Helix Core Server SDK package's .lib file.
2. Use *sigmake.exe* from the FLAIR SDK to create a .sig ("signature") file for all the .pat files from each given Perforce Helix Core Server SDK package.
3. Use *zipsig.exe* from the FLAIR SDK to compress each .sig file.
4. Disassemble Helix Core Server's *p4s.exe* file with IDA Pro and save the resulting .idb ("IDA database") file.
5. For each .sig file, open the .idb file, apply the .sig file, count the number of .sig file function matches, and close the .idb file without saving the modifications.
6. Rank the number of function matches for each .sig file.

After following the process above, we found the debug symbols from *p4api_vs2017_static_openssl1.1.1.zip* had the most function matches in *p4s.exe*:

Function Matches	Signature File
11,928	p4api_vs2017_static_openssl1.1.1_p4api-2023.1.2468153-vs2017_static.sig
11,887	p4api_vs2017_static_openssl3_p4api-2023.1.2468153-vs2017_static.sig

11,847	p4api_vs2017_static_openssl1.0.2_p4api-2023.1.2468153-vs2017_static.sig
11,847	p4api_vs2017_static_p4api-2023.1.2468153-vs2017_static.sig
10,228	p4api_vs2017_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2017_static_vsdebug.sig
10,187	p4api_vs2017_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2017_static_vsdebug.sig
10,147	p4api_vs2017_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2017_static_vsdebug.sig
10,147	p4api_vs2017_static_vsdebug_p4api-2023.1.2468153-vs2017_static_vsdebug.sig
8,222	p4api_vs2019_static_openssl1.1.1_p4api-2023.1.2468153-vs2019_static.sig
8,195	p4api_vs2019_static_openssl3_p4api-2023.1.2468153-vs2019_static.sig
8,167	p4api_vs2019_static_openssl1.0.2_p4api-2023.1.2468153-vs2019_static.sig
8,167	p4api_vs2019_static_p4api-2023.1.2468153-vs2019_static.sig
7,804	p4api_vs2019_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2019_static_vsdebug.sig
7,777	p4api_vs2019_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2019_static_vsdebug.sig
7,749	p4api_vs2019_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2019_static_vsdebug.sig
7,749	p4api_vs2019_static_vsdebug_p4api-2023.1.2468153-vs2019_static_vsdebug.sig
5,818	p4api_vs2022_static_openssl1.1.1_p4api-2023.1.2468153-vs2022_static.sig
5,802	p4api_vs2022_static_openssl3_p4api-2023.1.2468153-vs2022_static.sig

5,784	p4api_vs2022_static_openssl1.0.2_p4api-2023.1.2468153-vs2022_static.sig
5,784	p4api_vs2022_static_p4api-2023.1.2468153-vs2022_static.sig
5,525	p4api_vs2022_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2022_static_vsdebug.sig
5,509	p4api_vs2022_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2022_static_vsdebug.sig
5,491	p4api_vs2022_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2022_static_vsdebug.sig
5,491	p4api_vs2022_static_vsdebug_p4api-2023.1.2468153-vs2022_static_vsdebug.sig
1,639	p4api_vs2015_static_openssl1.1.1_p4api-2023.1.2468153-vs2015_static.sig
1,639	p4api_vs2015_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2015_static_vsdebug.sig
1,630	p4api_vs2015_static_openssl1.0.2_p4api-2023.1.2468153-vs2015_static.sig
1,630	p4api_vs2015_static_p4api-2023.1.2468153-vs2015_static.sig
1,630	p4api_vs2015_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2015_static_vsdebug.sig
1,630	p4api_vs2015_static_vsdebug_p4api-2023.1.2468153-vs2015_static_vsdebug.sig
1,628	p4api_vs2015_static_openssl3_p4api-2023.1.2468153-vs2015_static.sig
1,628	p4api_vs2015_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2015_static_vsdebug.sig
1,042	p4api_vs2013_static_openssl1.1.1_p4api-2023.1.2468153-vs2013_static.sig
1,041	p4api_vs2013_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2013_static_vsdebug.sig

1,040	p4api_vs2013_static_openssl1.0.2_p4api-2023.1.2468153-vs2013_static.sig
1,040	p4api_vs2013_static_p4api-2023.1.2468153-vs2013_static.sig
1,039	p4api_vs2013_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2013_static_vsdebug.sig
1,039	p4api_vs2013_static_vsdebug_p4api-2023.1.2468153-vs2013_static_vsdebug.sig
1,033	p4api_vs2013_static_openssl3_p4api-2023.1.2468153-vs2013_static.sig
1,032	p4api_vs2013_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2013_static_vsdebug.sig
973	p4api_vs2012_static_openssl1.1.1_p4api-2023.1.2468153-vs2012_static.sig
972	p4api_vs2012_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2012_static_vsdebug.sig
971	p4api_vs2012_static_openssl1.0.2_p4api-2023.1.2468153-vs2012_static.sig
971	p4api_vs2012_static_p4api-2023.1.2468153-vs2012_static.sig
970	p4api_vs2012_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2012_static_vsdebug.sig
967	p4api_vs2012_static_openssl3_p4api-2023.1.2468153-vs2012_static.sig
966	p4api_vs2012_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2012_static_vsdebug.sig
838	p4api_vs2010_static_openssl1.1.1_p4api-2023.1.2468153-vs2010_static.sig
838	p4api_vs2010_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2010_static_vsdebug.sig

837	p4api_vs2010_static_openssl1.0.2_p4api-2023.1.2468153-vs2010_static.sig
837	p4api_vs2010_static_p4api-2023.1.2468153-vs2010_static.sig
837	p4api_vs2010_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2010_static_vsdebug.sig
837	p4api_vs2010_static_vsdebug_p4api-2023.1.2468153-vs2010_static_vsdebug.sig
833	p4api_vs2010_static_openssl3_p4api-2023.1.2468153-vs2010_static.sig
833	p4api_vs2010_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2010_static_vsdebug.sig
495	p4api_vs2008_static_openssl1.1.1_p4api-2023.1.2468153-vs2008_static.sig
495	p4api_vs2008_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2008_static_vsdebug.sig
494	p4api_vs2008_static_openssl1.0.2_p4api-2023.1.2468153-vs2008_static.sig
494	p4api_vs2008_static_p4api-2023.1.2468153-vs2008_static.sig
494	p4api_vs2008_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2008_static_vsdebug.sig
494	p4api_vs2008_static_vsdebug_p4api-2023.1.2468153-vs2008_static_vsdebug.sig
490	p4api_vs2008_static_openssl3_p4api-2023.1.2468153-vs2008_static.sig
490	p4api_vs2008_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2008_static_vsdebug.sig
440	p4api_vs2005_static_openssl1.1.1_p4api-2023.1.2468153-vs2005_static.sig
440	p4api_vs2005_static_vsdebug_openssl1.1.1_p4api-2023.1.2468153-vs2005_static_vsdebug.sig

439	p4api_vs2005_static_openssl1.0.2_p4api-2023.1.2468153-vs2005_static.sig
439	p4api_vs2005_static_p4api-2023.1.2468153-vs2005_static.sig
439	p4api_vs2005_static_vsdebug_openssl1.0.2_p4api-2023.1.2468153-vs2005_static_vsdebug.sig
439	p4api_vs2005_static_vsdebug_p4api-2023.1.2468153-vs2005_static_vsdebug.sig
435	p4api_vs2005_static_openssl3_p4api-2023.1.2468153-vs2005_static.sig
435	p4api_vs2005_static_vsdebug_openssl3_p4api-2023.1.2468153-vs2005_static_vsdebug.sig

The remainder of this blog post leverages these signatures for *p4s.exe's* function names and type information.

Investigating the RPC header

Given that Helix Core Server runs as *LocalSystem*, local elevation of privilege attacks would certainly be worthwhile to explore. However, remote attacks via a network are much more intriguing from a vulnerability research perspective. Our next step is to investigate how Helix Core Server handles data it receives from remote users, or in our case, attackers.

Using TCPView, we can see that *p4s.exe* is listening for incoming connections on TCP port 1666:

Process Name	Protocol	State	Local Address	Local Port	Module Name
p4s.exe	TCP	Listen	0.0.0.0	1666	Perforce

Figure 6. TCPView showing Helix Core Server's listening TCP port

Programs built for Windows that listen on TCP ports for incoming connections almost always use Winsock's *recv()* function to receive incoming network data from clients. Using IDA Pro's cross-references ("CODE XREF"s below), we can see that *recv()* is called by several functions:

```
.idata:00007FF736BA9BA8 ; int (__stdcall *recv)(SOCKET s, char *buf, int len, int flags)
.idata:00007FF736BA9BA8                 extrn recv:qword        ; CODE XREF: nw_in_read+3A↑p
.idata:00007FF736BA9BA8                                         ; NetTcpTransport::Close(void)+14A↑p
.idata:00007FF736BA9BA8                                         ; NetTcpTransport::IsAlive(void)+62↑p
.idata:00007FF736BA9BA8                                         ; NetTcpTransport::Peek(int,char *,int)+4B↑p
.idata:00007FF736BA9BA8                                         ; NetTcpTransport::SendOrReceive(NetIoPtrs &,Error *,Error *)+62↑p
.idata:00007FF736BA9BA8                                         ; NetTcpTransport::SendOrReceive(NetIoPtrs &,Error *,Error *)+4EC↑p
.idata:00007FF736BA9BA8                                         ; NetTcpTransport::SendOrReceive(NetIoPtrs &,Error *,Error *)+589↑p
.idata:00007FF736BA9BA8                                         ; NetSslTransport::Close(void)+153↑p
.idata:00007FF736BA9BA8                                         ; NetSslTransport::Close(void)+3FA↑p
.idata:00007FF736BA9BA8                                         ; sub_7FF736A593A0+3B↑p
```

Figure 7. Code cross-references to recv()

We're looking to assess how received network data is parsed and handled, and to save time in determining which of the functions above actually receives the connected client data via *recv()*, we used a debugger to set a breakpoint on *recv()* and reviewed its thread's call-stack to reveal the following chain of function calls:

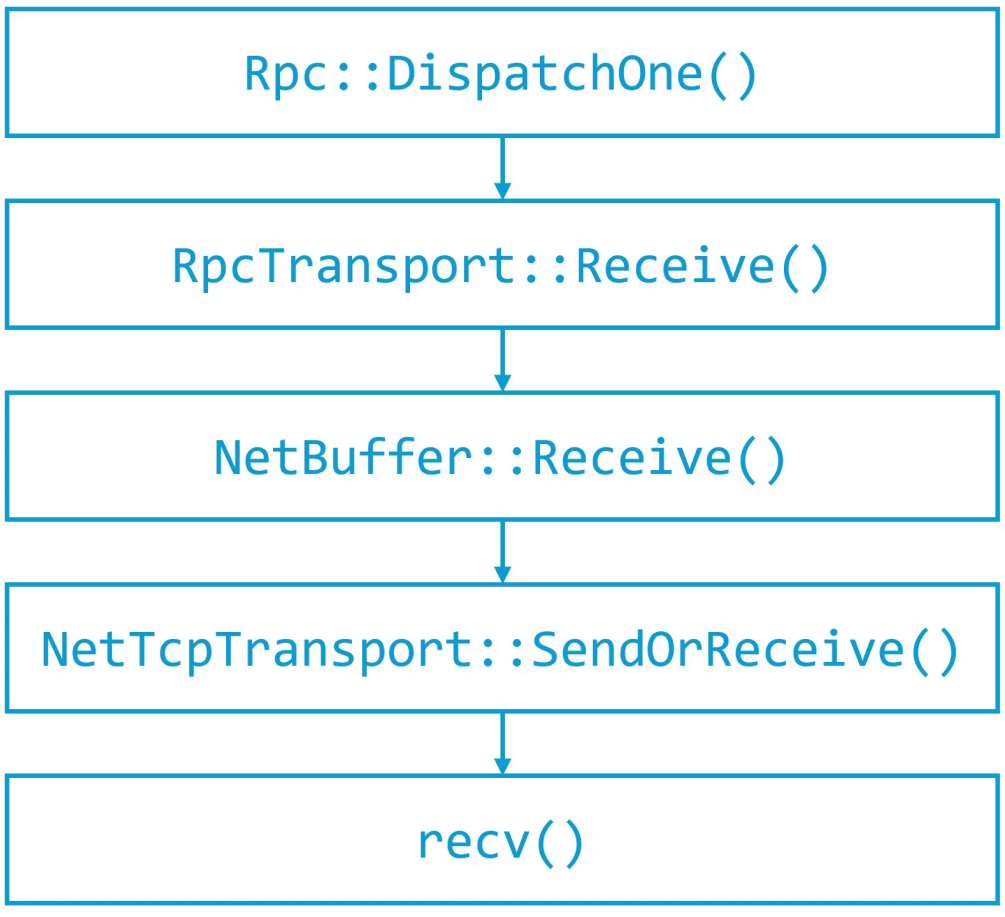


Figure 8. The function call-stack for recv() at runtime

In the call-stack above, "*Rpc*" is short for "Remote Procedure Call", a common term used for remotely executing functions.

Although we're assessing the Helix Core Server, the function *RpcTransport::Receive()* (in Figure 8) is also included in the client source code discussed above (note that the comments are from Perforce's developers, not from Microsoft):

```
61  NO_SANITIZE_UNDEFINED
62  int
63  RpcTransport::Receive( StrBuf *s, Error *re, Error *se )
64  {
65      // Get the five byte length header.
66
67      unsigned char l[5];
68
69      if( !( NetBuffer::Receive( (char *)l, 5, re, se ) ) )
70          return 0;
71
72      if( l[0] != ( l[1] ^ l[2] ^ l[3] ^ l[4] ) )
73      {
74          re->Set( MsgRpc::NotP4 );
75          return -1;
76      }
77
78      int length =
79          l[1] * 0x1 +
80          l[2] * 0x100 +
81          l[3] * 0x10000 +
82          l[4] * 0x1000000;
83
84      // Lengths < 11 are not enough for a func variable... bzzzz...
85      if( length < 11 || length >= 0x1fffffff )
86      {
87          re->Set( MsgRpc::NotP4 );
88          return -1;
89      }
90
91      // Now allocate the buffer and read the data.
92
93      if( !( NetBuffer::Receive( s->Alloc( length ), length, re, se ) ) )
94      {
95          re->Set( MsgRpc::Read );
96          return -1;
97      }
98
99      return 1;
100 }
```

Figure 9. Source code for RpcTransport::Receive()

The code above does the following:

1. On line 69, calls *NetBuffer::Receive()* to receive five bytes of data from the connected TCP client. We will refer to these five bytes as the RPC header.
2. On line 72, verifies that the first byte's value equals the value of the following four bytes using the XOR operation to compute a parity byte checksum.
3. On line 78, interprets those following four bytes as a 32-bit little-endian value named *length*.
4. On line 85, verifies that *length* >= *12* and that *length* < *0x1FFFFFFF*.
5. On line 93, allocates memory of size *length* and receives *length* bytes from the connected TCP client.

However, there's a design risk in the code above, in that there's not sufficient protection against asymmetric resource consumption attacks from remote unauthenticated attackers. An attacker could connect to the Helix Core Server, send a five-byte RPC header specifying a *length* value of *0x1FFFFFFE*, and cause the server to allocate *0x1FFFFFFE* bytes (about 537 MB) of memory. An attacker could exploit this vulnerability by establishing numerous connections and requesting these large memory allocations via each connection, quickly consuming all the server's available memory. Once available memory is exhausted, the next call to *Alloc()* (step 5 above) will lead Helix Core Server's memory allocator (which happens to be mimalloc) to throw an unhandled *std::bad_alloc()* exception from *mi_try_new_handler()*, causing the Helix Core Server process to crash and not restart. This denial-of-service (DoS) attack is exploitable by remote unauthenticated attackers.

This vulnerability is now identified as CVE-2023-5759 and it has a CVSS score of 7.5.

Investigating RPC handler functions

We showed in the call-stack above that *RpcTransport::Receive()* is called by *Rpc::DispatchOne()*. This latter function takes the allocated buffer received by *RpcTransport::Receive()*, parses it as an RPC command with optional arguments, looks up the handler for the given RPC command, and calls the handler with the received arguments. Many of these RPC commands are mapped to the p4 commands listed here. Specifically, there are 202 formally documented p4 commands, and about 450 defined RPC commands, though not all RPC commands have their handlers registered by default at runtime.

Since we're most interested in the possibility of remote unprivileged attacks against Helix Core Server in its default configuration, we created our own Perforce client from scratch that attempts to call (without any authentication) each of the approximately 450 RPC commands defined in *p4s.exe*. Of those, we found that about 360 RPC commands have their handlers registered by default at runtime. This is too high of a count to manually assess in a reasonable amount of time, so we had to find other means to prioritize our RPC command analysis.

We found that *p4s.exe* statically imports 382 API functions. Of those, we identified the most interesting functions that could potentially achieve remote code execution, assuming an unauthenticated remote attacker could both execute an RPC function that calls one of these API functions and control the arguments to that API function. These functions are:

- *CreateFileMapping()*
- *CreateProcess()*
- *DeviceIoControl()*
- *LoadLibrary()*
- *LoadLibraryEx()*
- *MapViewOfFile()*
- *MoveFileEx()*
- *RegSetValueEx()*
- *ShellExecuteEx()*
- *WriteFile()*

Assessing this short list of API functions and analyzing code-flow paths from RPC handlers to these functions was a much more tractable problem than manually reviewing each of the approximately 360 registered RPC handlers.

The bgtask command

By reviewing cross-references with IDA Pro, we were able to identify the following call-chain from an RPC command handler to *CreateProcess()*:

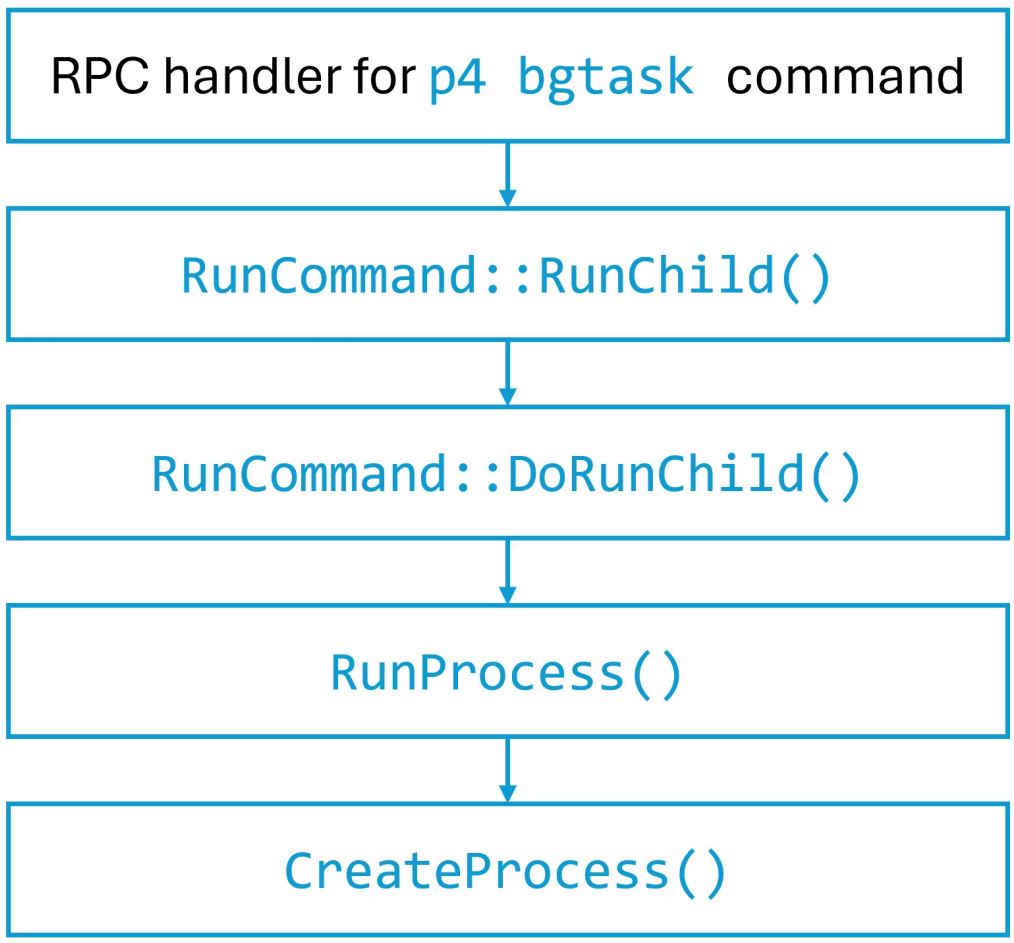


Figure 10. The function call-chain from bgtask to CreateProcess()

According to Perforce's documentation, the *p4 bgtask* command "enables a Helix Core superuser on the p4 command-line client to run commands or programs remotely on the server in the background." It's thus not surprising that this type of command would end up calling *CreateProcess()*, but since the documentation states that this command can only be run by a superuser, our only hope of finding a security vulnerability here was if there was a bug in the authentication component or in how the RPC arguments were getting parsed.

To begin our assessment of *p4 bgtask*, we used the custom Perforce client that we wrote to see how the server would respond if we tried remotely calling *bgtask* without any authentication. To our surprise, the server didn't return any errors. In fact, the server ran the command line that we sent to it, and this child process ran as *LocalSystem*.

Upon further investigation, this is by design, with the manual noting to "Run *p4 protect* immediately after installing Helix Server for the first time. Before the first call to *p4 protect*, every Helix Server user is a superuser and thus can access and change anything in the depot". In this context, "every Helix Server user" also includes unauthenticated anonymous remote users.

If an administrator does not manually perform those post-installation steps, this missing authentication for a critical function allows unauthenticated remote attackers to run arbitrary command lines (including PowerShell command lines with script blocks) as *LocalSystem* when Helix Core Server is installed with its default configuration.

This vulnerability is now identified as CVE-2023-45849 and it has a CVSS score of 10.0.

The rmt-Shutdown RPC handler

When a user (or attacker) uses the *p4 bgtask* discussed above with the standard Perforce Client, the client sends the RPC command name *user-bgtask* to the server to execute that command. However, some RPC command names that are accepted by the server don't have a corresponding Perforce Client command; one of those RPC command names is *rmt-Shutdown*.

Although it's not possible to send the *rmt-Shutdown* RPC command with the standard Perforce Client (nor the Perforce Admin Tool), and it doesn't appear to be documented on Perforce's website, we were able to send the command with our custom Perforce client. We found that the handler for *rmt-Shutdown* requires a username of *remote* but doesn't require any authentication credentials for that username. When the Perforce Helix Core Server receives this command, it terminates the Helix Core Server process, thereby allowing unauthenticated remote attackers to perform DoS attacks against the server.

This vulnerability is now identified as CVE-2023-35767 and it has a CVSS score of 7.5.

The rmt-UpdtFovrCommit RPC handler

Similar to the *rmt-Shutdown* RPC command name, the RPC command name *rmt-UpdtFovrCommit* (which is likely short for "remote update failover commit") cannot be sent via the standard Perforce Client nor Admin Tool and doesn't appear to be documented on Perforce's website but can be sent with a custom Perforce client. This RPC function piqued our interest when we first tested for registered RPC handlers, since when our custom-built scanner sent a *rmt-UpdtFovrCommit* RPC command as an anonymous user and without any command arguments to the Helix Core Server, the Helix Core Server process crashed.

We can see the reason for the crash in the decompiled *rmt-UpdtFovrCommit* handler code below:

```
pStrPtrUser = StrDict::GetVar(pRpc, "user");
pStrPtrMid = StrDict::GetVar(pRpc, "mID", pError);
pStrPtrSid = StrDict::GetVar(pRpc, "sID", pError);
pStrPtrFailoverSeen = StrDict::GetVar(pRpc, "failoverSeen");
pStrPtrNonMandatory = StrDict::GetVar(pRpc, "notMandatory");
pStrPtrMandatory = StrDict::GetVar(pRpc, "mandatory");
pStrPtrMasterAddress = StrDict::GetVar(pRpc, "masterAddress");
pStrPtrStandbyAddress = StrDict::GetVar(pRpc, "standbyAddress");
strBuf.size = 0;
strBuf.length = 0;
strBuf.buffer = g_szEmptyString2;
strBufUser.size = 0;
strBufUser.length = 0;
strBufUser.buffer = g_szEmptyString2;
StrOps::PackOctet(&strBufUser, pStrPtrUser);
```

Figure 11. Snippet of code from the decompiled rmt-UpdtFovrCommit handler function

As previously discussed, RPC messages sent from the client to the server contain the RPC function name and can optionally contain RPC function arguments. In the code above, *StrDict::GetVar()* is used to get the client's RPC function arguments from the *pRpc* object. If the given argument name was not provided in the client's RPC message, then *StrDict::GetVar()* returns zero. In the first line above, *StrDict::GetVar()* is used to get the value of the *user* RPC function argument. However, if the user (or attacker) does not specify a value for *user* in their RPC message then *pStrPtrUser* gets set to zero. In the last line above, we see *pStrPtrUser* passed as the second argument to *StrOps::PackOctet()*, (the source code for which is available in the client source code discussed above):

```
101   class StrPtr {
      ...
115   p4size_t    Length() const
116       { return length; }
      ...
268       protected:
269       char    *buffer;
270       p4size_t    length;
      ...
307   } ;
```

Figure 12. Source code snippets from strbuf.h

```
475  void
476  StrBuf::Append( const StrPtr *t )
477  {
478      char *s = Alloc( t->Length() + 1 );
479      memmove( s, t->Text(), t->Length() );
480      s[ t->Length() ] = '\0';
481      --length;
482  }
```

Figure 13. Source code snippet from strbuf.cc

```
1099  void
1100  StrOps::PackOctet( StrBuf &o, const StrPtr &s )
1101  {
1102      o.Append( &s );
1103  }
```

Figure 14. Source code snippet from strops.cc

As can be seen in the code above, when *StrOps::PackOctet()* is called with zero as the value for *s*, *StrBuf::Append()* gets called with zero as the value for *t*. This results in *StrBuf::Append()* trying to dereference the *length* field of *t*, where the *buffer* field of *t* is at offset *0* relative to the beginning of the *t* object and the *length* field of *t* is at offset *8* relative to the beginning of the *t* object (since *char *buffer* is 64-bits). When the value of *t* is *0*, dereferencing *length* leads to reading from virtual memory address *0x0000000000000008*, which results in a read-access violation or segmentation fault. We found that these types of exceptions are not handled gracefully by the server and that such read-exceptions cause the entire server process to crash and not restart. This DoS attack is exploitable by remote unauthenticated attackers.

This vulnerability is now identified as CVE-2023-45319 and it has a CVSS score of 7.5.

Coordinated disclosure

Microsoft reported these four security vulnerabilities to the vendor Perforce at the end of August 2023. Immediately afterwards, on September 1, Perforce acknowledged these four vulnerabilities and began work to investigate and remediate them. Throughout September and October, Perforce communicated status updates to Microsoft on implementing fixes and putting those fixes through their QA processes. Perforce reserved CVE IDs on October 24, 2023, shared those IDs with Microsoft on October 25, 2023, and informed Microsoft at that time that the patches would be published by mid-November 2023. On November 7, 2023, Perforce published Perforce Helix Core Server version 2023.1/2513900, which mitigates these four vulnerabilities. Perforce has also published patches for the older Perforce Helix Core Server versions 2022.2, 2022.1, and 2021.2. Perforce recommends upgrading to the latest Perforce Helix Core Server version 2023.2.

Microsoft would like to thank Perforce for their professionalism and for their rapid response in addressing these security vulnerabilities. Microsoft is grateful for this partnership and for Perforce's commitment to security.

Mitigation and protection guidance

Microsoft is not aware of any adversaries exploiting these vulnerabilities, but mitigations should be applied by all Helix Core Server customers as soon as possible.

Risk detection

Extend vulnerability and risk detection beyond the firewall with platforms like Microsoft Defender External Attack Surface Management. Customers can identify internet-exposed infrastructure running Perforce Helix Core Server in their inventory and use the insights tile under the Attack Surface Summary dashboard to surface assets vulnerable to CVE-2023-5759, CVE-2023-45849, CVE-2023-35767, and CVE-2023-45319.

What to do now if you're affected

Update your Perforce Helix Core Server: https://www.perforce.com/downloads/helix-core-p4d.

Defense-in-depth

In addition to following Perforce's guidance on "Securing the server", Microsoft recommends adhering to the following defense-in-depth tactics to minimize the risk of exploitation of these or other Perforce Helix Core Server vulnerabilities.

- Regularly monitor for and apply patches for third-party software.
- Use a VPN and/or an IP allow-list to limit who can communicate with your Perforce Helix Core Server.
- Issue TLS certificates to legitimate Perforce users and use a TLS termination proxy in front of Perforce Helix Core Server to validate client's TLS certificates before allowing them to connect to Perforce Server.
- Log all access to your Helix Core Server, both via your network appliances and via Perforce Helix Core Server itself.
- Configure alerting to notify IT administrators and your security team if the Helix Core Server process crashes.
- Use network segmentation to ensure that if your Helix Core Server is compromised, an attacker's ability to pivot in your network is limited.

Appendix

Threat intelligence reports

Microsoft customers can use the following reports in Microsoft products to get the most up-to-date information about the threat actor, malicious activity, and techniques discussed in this blog. These reports provide intelligence, protection information, and recommended actions to prevent, mitigate, or respond to associated threats found in customer environments.

Microsoft Defender Threat Intelligence

- CVE-2023-5759
- CVE-2023-45849
- CVE-2023-35767
- CVE-2023-45319

Microsoft 365 Defender Threat analytics

- Vulnerability profile: Critical remote code execution vulnerability in Perforce Helix Core Server

Jason Geffner

Microsoft Threat Intelligence Community

References

- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-5759
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-35767
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-45319
- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-45849
- https://ftp.perforce.com/perforce/r23.1/bin.ntx64/

- https://portal.perforce.com/s/article/3925
- https://swarm.workshop.perforce.com/projects/perforce_software-p4
- https://www.perforce.com/downloads/administration-tool
- https://www.perforce.com/downloads/helix-command-line-client-p4
- https://www.perforce.com/downloads/helix-core-c/c-api
- https://www.perforce.com/downloads/helix-core-p4d
- https://www.perforce.com/manuals/cmdref/Content/CmdRef/commands.html
- https://www.perforce.com/manuals/p4sag/Content/P4SAG/protections.when_to_set.html
- https://www.perforce.com/press-releases/perforce-open-sources-popular-version-control-tools

Acknowledgments

Microsoft would like to recognize https://www.keysight.com/blogs/tech/nwvs/2022/06/08/a-sneak-peek-into-the-protocol-behind-perforce for previous work done in analyzing Perforce's RPC protocol.

Learn more

For the latest security research from the Microsoft Threat Intelligence community, check out the Microsoft Threat Intelligence Blog: https://aka.ms/threatintelblog.

To get notified about new publications and to join discussions on social media, follow us on LinkedIn at https://www.linkedin.com/showcase/microsoft-threat-intelligence, and on Twitter at https://twitter.com/MsftSecIntel.

To hear stories and insights from the Microsoft Threat Intelligence community about the ever-evolving threat landscape, listen to the Microsoft Threat Intelligence podcast: https://thecyberwire.com/podcasts/microsoft-threat-intelligence.

Related Posts

Research Threat intelligence
Microsoft Defender Vulnerability Management
Vulnerabilities and exploits
Sep 14, 2023 13 min read

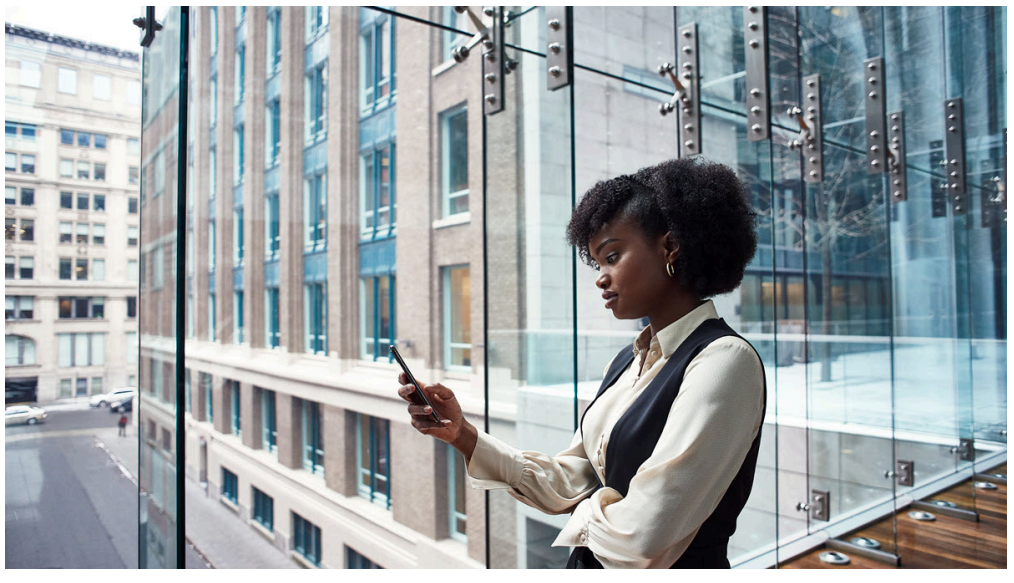
Uncursing the ncurses: Memory corruption vulnerabilities found in library ›

A set of memory corruption vulnerabilities in the ncurses library could have allowed attackers to chain the vulnerabilities to elevate privileges and run code in the targeted program's context or perform other malicious actions.

Research Threat intelligence Microsoft Defender for IoT
Vulnerabilities and exploits
Aug 10, 2023 10 min read

Multiple high severity vulnerabilities in CODESYS V3 SDK could lead to RCE or DoS ›

Microsoft researchers identified multiple high-severity vulnerabilities in the CODESYS V3 SDK that could put operational technology (OT) infrastructure at risk of attacks, such as remote code execution (RCE) and denial of service (DoS).



Research Threat intelligence Microsoft Defender for Endpoint Vulnerabilities and exploits

May 30, 2023 10 min read

New macOS vulnerability, Migraine, could bypass System Integrity Protection ›

A new vulnerability, which we refer to as "Migraine", could allow an attacker with root access to bypass System Integrity Protection (SIP) in macOS and perform arbitrary operations on a device.

Research Threat intelligence Vulnerabilities and exploits

Mar 6, 2023 6 min read

Protecting Android clipboard content from unintended exposure ›

Microsoft discovered that the SHEIN Android application periodically read the contents of the Android device clipboard and, if a particular pattern was present, sent the contents of the clipboard to a remote server.

Get started with Microsoft Security

Microsoft is a leader in cybersecurity, and we embrace our responsibility to make the world a safer place.

Learn more

Connect with us on social