
WHITE PAPER

# The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks.

by Ben Seri & Gregory Vishnepolsky

# Armis Labs

Armis Labs revealed a new attack vector endangering major mobile, desktop, and IoT operating systems, including Android, iOS, Windows, and Linux, and the devices using them. The new vector is dubbed “BlueBorne”, as it spreads via the air and attacks devices via Bluetooth. BlueBorne allows attackers to take control of devices, access corporate data and networks, penetrate secure “air-gapped” networks, and spread malware to other devices. The attack does not require the targeted device  to be set on discoverable mode or to be paired to the attacker’s device. In addition, the targeted user is not required to authorize or authenticate the connection to the attacker’s device.

Armis Labs has identified eight vulnerabilities which can be used as part of the attack vector so far. These vulnerabilities are fully operational, and were successfully turned into exploits, as we will demonstrate in future blog posts. These are the vulnerabilities:

1. Linux kernel RCE vulnerability - CVE-2017-1000251
2. Linux Bluetooth stack (BlueZ) information Leak vulnerability - CVE-2017-1000250
3. Android information Leak vulnerability - CVE-2017-0785
4. Android RCE vulnerability #1 - CVE-2017-0781
5. Android RCE vulnerability #2 - CVE-2017-0782
6. The Bluetooth Pineapple in Android - Logical Flaw CVE-2017-0783
7. The Bluetooth Pineapple in Windows - Logical Flaw CVE-2017-8628
8. Apple Low Energy Audio Protocol RCE vulnerability - CVE-2017-14315

This research paper explores the attack surface around each of the vulnerabilities, explaining the areas in Bluetooth’s implementations in which they were found. In addition, it provides a detailed explanation of the internal workings and an impact analysis of each vulnerability. We hope our research will encourage and help others to audit other Bluetooth stacks, and reveal additional weak spots in the major implementations of Bluetooth stacks.

## Acknowledgment

We would like to thank Alon Livne for the development of the Linux RCE exploit.

## Introduction to Bluetooth

Bluetooth is the leading and most widespread protocol for short-range communications. According to estimates, more than 8.2 billion Bluetooth devices are currently in use, and the number grows by the day. Bluetooth is implemented in a very wide range of devices, from the most popular consumer products (Smartphones, Wearables), to the most common appliances in enterprises (PCs, Smart TVs, Printers), and even in the critical infrastructure of our lives - medical appliances, cars, and many more. Bluetooth is managed, licensed and maintained by the Bluetooth Special Interests Group (SIG), which includes members from several large technology companies such as Microsoft, Intel, Apple, IBM, and more.

Though it was first introduced to the world in 1998, Bluetooth continues to develop with BLE and Mesh topology as the most interesting examples. BLE (Bluetooth Low Energy) is the cool new variant of Bluetooth, and is rapidly gaining ground in the market as it allows a new generation of devices, such as “smart” sensors and remote controls, which have limited power supply and bandwidth to connect to existing Bluetooth devices such as smartphones and PCs. Aside from BLE, a new feature was introduced in Bluetooth 5.0 - Bluetooth Mesh. This new feature changes the topology of Bluetooth connections by allowing low level devices to interconnect and form larger networks with a more elaborate and dense structure. The linked nature of the Mesh topology enables a Bluetooth network to spread far and wide and allow devices on the far ends of it to communicate. This new feature is an attempt by the Bluetooth SIG to compete with other rising short-range wireless protocols (like Zigbee, Z-Wave, LoRa and others) in handling the ever expanding realm of smart IoT devices and its unique requirements.

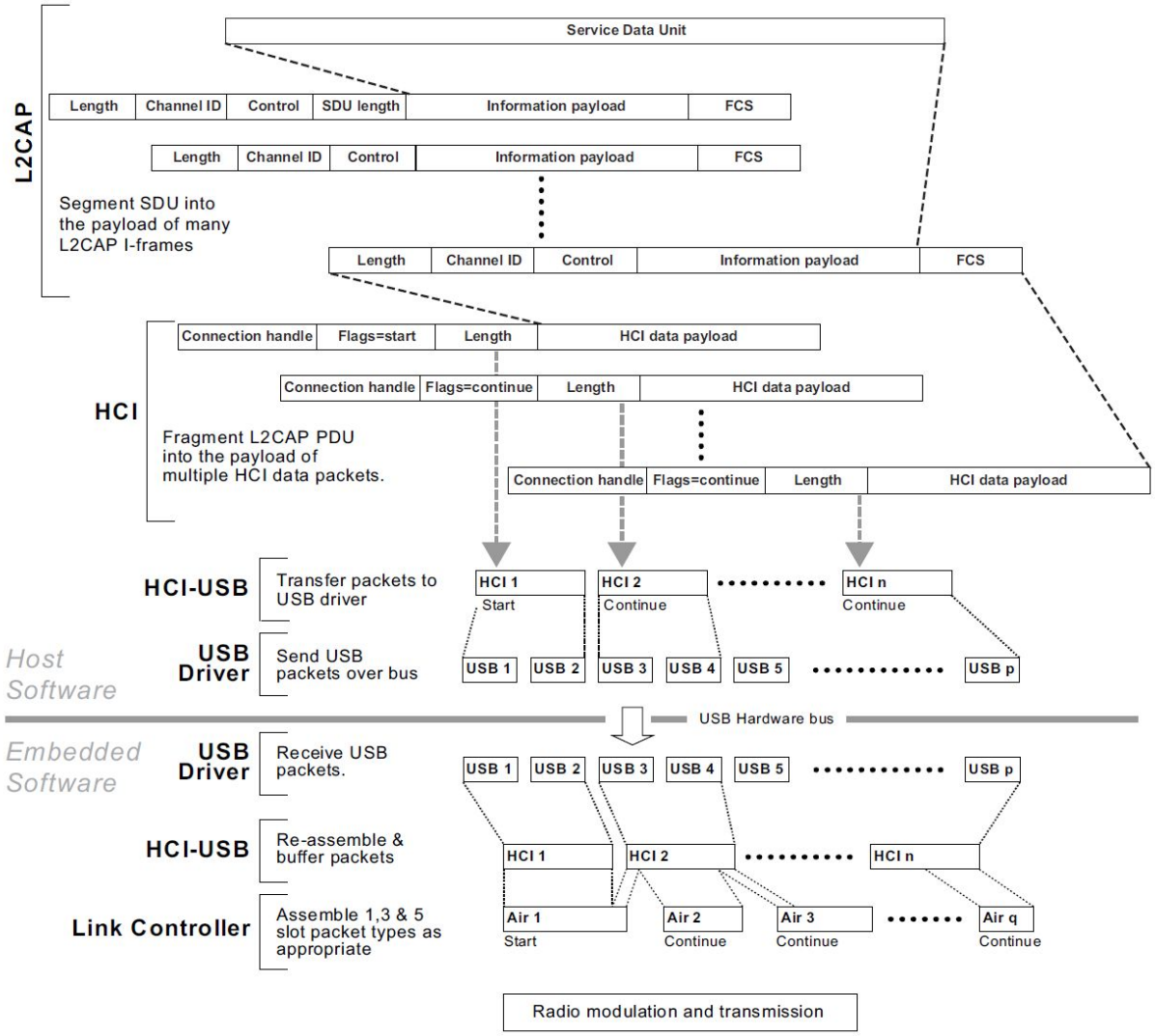
The recent developments in Bluetooth, together with its long history, are what make this protocol the backbone of short range connectivity in the vast majority of devices in the market. The growing reliance on wireless connectivity throughout our lives is likely to turn this protocol into an even bigger part of them, and of the devices we use.

## So, what seems to be the problem?

Bluetooth is complicated. Too complicated. Too many specific applications are defined in the stack layer, with endless replication of facilities and features. These over-complications are a direct result of the immense work, and over-engineering that was put into creating the Bluetooth specification. Just to illustrate this point: while the WiFi specification (802.11) is only 450 pages long, the Bluetooth specification reaches 2822 pages.

Bluetooth’s complexity kept researchers from auditing its implementations at the same level of scrutiny that other highly exposed protocols, and outwards-facing interfaces have been treated with. The result of the lack of review is a large number of vulnerabilities, such as those which we are disclosing here. The complications in the specifications translate into multiple pitfall junctions in the various implementations of the Bluetooth standard.

An example of the unnecessary complexity of Bluetooth is fragmentation, a common concept in many protocols, and a soft spot in every implementation. The Bluetooth specification has no less than 4 different fragmentation layers implemented throughout the stack, as illustrated in this diagram taken from the specification:




Bluetooth Core Specification v5.0, Vol 3, Part A, Section 7.3.3, Page 1831

Aside from the fragmentation layer that only exists inside the the host machine (The USB Driver layer), from the radio layer (Link Controller) to the L2CAP layer Bluetooth has a total of 3 fragmentation layers (and additional fragmentation layers exist in some of Bluetooth's services as well):

- Air packets fragmentation in the Link Controller;
- HCI layer fragmentation (ACL level continuation);
- L2CAP segmentation.

The absurdity goes even further, as in some Bluetooth's services, a fragmentation mechanism can be spotted in every one of Bluetooth's layers along the way. Such is the case of SDP - a

packet will be fragmented by the SDP continuation mechanism, and then by L2CAP's segmentation mechanism, and then again by ACL continuation, and one last time by the fragmentation mechanism done the Link Controller.

## Past research of Bluetooth

Previous works focused on finding potential issues in the Bluetooth specification itself, showing the weakness of the encryption key exchange procedures in Bluetooth versions up to v2.1. Once Bluetooth introduced the “Secure Simple Pairing” - a feature that fixed many of the known pairing issues in the specification, the focus of the security community shifted away from Bluetooth. In more recent years, Bluetooth Low Energy emerged, causing renewed interest of the community in Bluetooth as a whole. That said, a thorough inspection of the various implementations of the Bluetooth stacks hasn’t been performed.

This work is an initial step in revealing the potential flaws in Bluetooth stacks. However, as the Bluetooth stack is such an immense piece of code, the work we are presenting might be only the tip of the iceberg.

## Demystifying Discoverability

Bluetooth is turned on by default on many devices, and most users prefer to leave it on since it is a convenient way of connecting headphones, keyboards, and other various IoT devices over the same familiar interface of the OS. Different types of Bluetooth connections exist, one of which is pairing between them.

In most OSs, when the user is actively trying to pair to a device, his machine is discoverable over Bluetooth by nearby peers. In any other case, discoverability is disabled. However, a Bluetooth enabled device is almost always listening for unicast traffic targeted to it, even when it is not set on discoverable mode (this is called “Page scan mode”). For this reason, to establish a connection the initiating party only needs to know the BDADDR (Bluetooth device address, MAC address) of the target device. Once an attacker acquires it, and is in physical proximity of the device (RF range) he or she can reach the surprisingly wide attack surface of its listening Bluetooth services.

Discovering BDADDRs of non-discoverable devices is considered difficult by some, including the specification itself which describes it as one of the "Four different entities are used for maintaining security at the link layer" (Bluetooth Specification Core v5.0, page 1649). The assumed difficulty arises due to the complexity of the Bluetooth protocol at the lower layers and the assumed lack of hardware capable to do so by “sniffing” the air. However, it is very easy to discover the BDADDRs, even of non-discoverable devices.

Open source hardware like the Ubertooth has been available for a number of years. This tool allows researchers to sniff and monitor the protocol in the physical and link layers (by sniffing the air for Bluetooth packets). Since the “Monitor Mode” of Bluetooth is very limited in tools widely

accessible for researchers, the introduction of Ubertooth reduced the barrier of entry for many (ourselves included).

Even though Bluetooth connections are encrypted, the packet headers (which are plaintext) contain enough information from which the BDADDRs of communicating devices can be derived. If a machine generates any Bluetooth traffic, an attacker in physical proximity can derive its BDADDR and use it to send unicast traffic to the device.

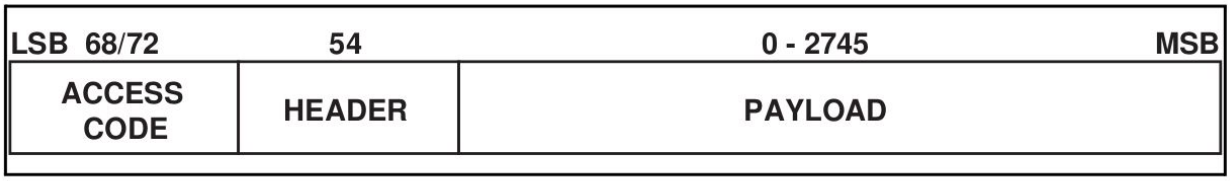


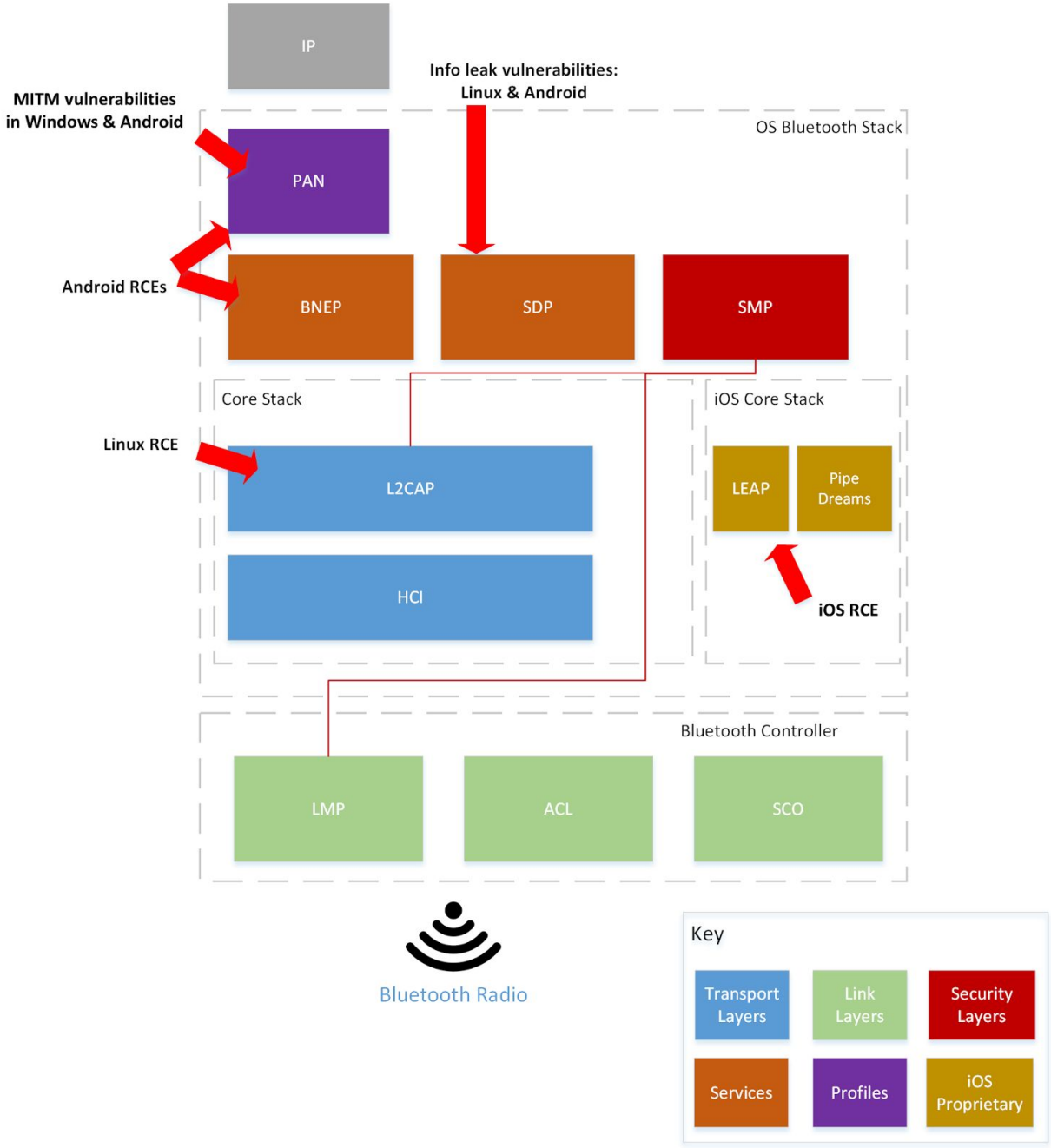

*Figure 6.1: General Basic Rate packet format.*

Structure of a Bluetooth classic packet in the air. The “Access Code” contains the 24-bit LAP part of a BDADDR

If the device generates no Bluetooth traffic, and is only listening, it is still possible to “guess” the BDADDR, by sniffing its WiFi traffic. This is viable since WiFi MAC addresses appear unencrypted over the air and due to the widely accepted norm of OEMs and hardware manufacturers that the MACs of internal Bluetooth/WiFi adapters are either the same, or only differ in the last digit (one being +1 of the other).

## Attack Surface Analysis

Having established the relative simplicity of obtaining the Bluetooth address of a device, we can now dive into the wide attack surface that exists in every Bluetooth stack, throughout the protocol’s layers. We will review the Bluetooth layers, from L2CAP, to SMP, to SDP, and then to the higher layers we’ve examined: BNEP, and PAN. In each layer we will describe our findings and explain the vulnerabilities we’ve disclosed.




Basic blocks in the Bluetooth stack, indicating the location of various vulnerabilities

## Widespread Bluetooth Stacks

In some sense, a Bluetooth stack is the equivalent of TCP/IP stack, only for Bluetooth communications. Unlike other low level communication protocols such as Ethernet, WiFi, and 6LoWPAN, Bluetooth does not rely on TCP/IP stack for all the high level applications protocols. Instead, a wide range of protocols and applications were defined by the Bluetooth SIG, and are referred to collectively as the Bluetooth Stack.

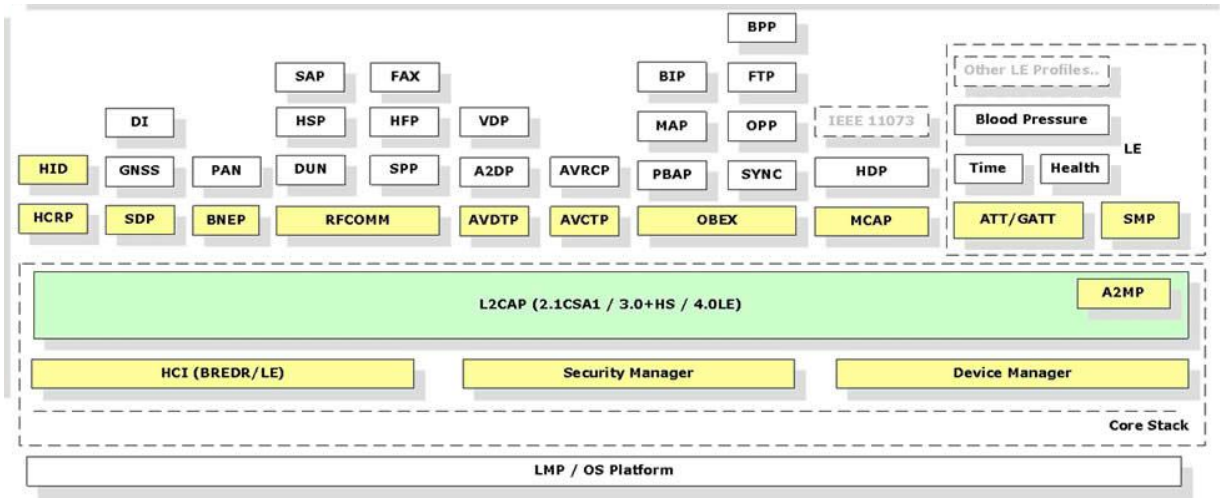



Image X - The Bluetooth stack architecture

The Bluetooth stack constitutes a full alternative to the classic 7 layer stack of TCP/IP - starting at the physical layer, and spanning up to the application layer. The lower layers of the stack - the physical and link layers - are implemented in Bluetooth chips. These chips communicate with a “Host”, which is the actual operating system of the device, through Bluetooth’s HCI (Host-Controller interface) protocol. All protocols above this layer (such as L2CAP, AMP, SMP, SDP, and RFCOMM) are implemented on the host’s side. Each modern operating systems has only one Bluetooth stack, unlike drivers of network adapters which have different versions for each hardware piece. This means that any vulnerability found in one of these stacks automatically affects all devices running that specific OS, endangering numerous devices in the market.

The first significant stack is Linux’s BlueZ stack, which was used by early Android versions, and is still in use by Linux and other OSs derived from it, such as Samsung’s Tizen OS. Later on, Android developed its own stack, called Bluedroid or Fluoride, used in all Android devices from version 4.2 onwards. Windows has its own Bluetooth stack, since Windows XP, and Apple created two variations of the Bluetooth stack, one for iOS, and the other for OSX.

## L2CAP

### Overview

On the host side, the lowest layer in the Bluetooth stack is L2CAP. This layer is responsible for managing connections to the various Bluetooth services. The underlying transport of L2CAP is ACL - Asynchronous Connection-Oriented Logical transport. ACL is simply the packet-oriented, unreliable transport layer over which almost all Bluetooth data is transmitted. L2CAP manages connection-oriented channels over ACL, which are logical end-to-end transports identified by Channel IDs in the packet’s body. The role of those Channel IDs can be compared to the port used in TCP (or UDP) applications, and in general L2CAP can be seen as Bluetooth’s equivalent of TCP, as it also implements QoS and flow-control features. L2CAP also implements (yet another) fragmentation and reassembly mechanisms - and thus enables transport of large SDUs (Service

Data Units - L2CAP lingo for “large packets” - used by the various *services* over L2CAP).

The Bluetooth specification reserves specific CIDs (Channel IDs) for fixed purposes - as an example, CID 1 will always refer to the signaling channel in which control packets are passed (and through them - new connections can be established). Other CIDs are managed and allocated dynamically. The various services over Bluetooth often have fixed PSMs (Protocol/Service Multiplexer - another L2CAP term meaning port number), and an endpoint wishing to connect to these services would send an L2CAP ConnectionRequest message to that specific PSM. In response to this message a dynamic CID would be allocated to identify the connection to that specific service.

When creating a new L2CAP connection, the two endpoints attempt to coordinate an agreed upon configuration by passing packets called configuration requests and configuration responses back and forth. A configuration request contains several elements which determine the exact type of connection features which will be used.

### Mutual configuration

The configuration process takes place using configuration requests and responses, referred to in the specification as *L2CAP_ConfReq* and *L2CAP_ConfResp* messages. These messages are passed on the signaling channel, with both endpoints dispatching configuration requests to one another as part of the initial handshake, and replying with configuration responses. The configuration response contains a status code which informs the initiator whether his configuration was accepted or rejected. Each endpoint negotiates its own configuration, meaning the configuration parameters of both endpoints need to be agreed upon.

Figure A.1 illustrates the basic configuration process. In this example, the devices exchange MTU information. All other values are assumed to be default.

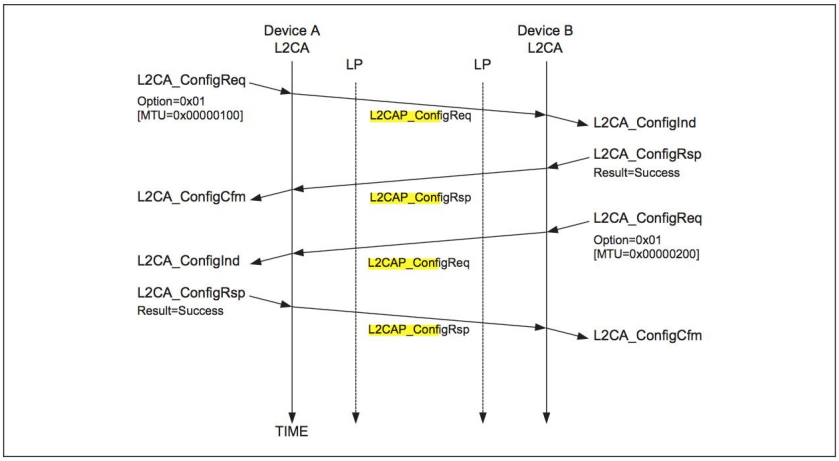



*Figure A.1: Basic MTU exchange*

Excerpt from Bluetooth Spec, page 1902

In the example above , Device A requests a Maximum Transmission Unit (**MTU**) of 0x100, which Device B accepts, followed by a request from Device B for an MTU of 0x200, which Device A

accepts as well. Two MTU parameters were agreed upon in this transaction - the maximum message size of outgoing messages from Device A to Device B is 0x100, and the the maximum message size of outgoing messages from Device B to Device A is 0x200.

While the above example is a simple exchange of parameters, a device might also choose to reject an offered configuration request due to “unacceptable parameters”. To ease re-negotiation, its configuration response may contain an alternative, acceptable value for the parameter it wishes to change. For example, in the following code-snippet (from BlueZ, the Linux Bluetooth stack), the requested MTU value is checked against a minimum value (chan->omtu is initiated to a default when the connection is established):

```
if (mtu < L2CAP_DEFAULT_MIN_MTU)
    result = L2CAP_CONF_UNACCEPT;
else {
    chan->omtu = mtu;
    set_bit(CONF_MTU_DONE, &chan->conf_state);
}
l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->omtu);
```

Excerpt from net/bluetooth/l2cap_core.c

If the requested MTU value is valid, it is committed to the current connection settings and the MTU configuration state is marked as "done" in the channel object, otherwise, the reply value is set to UNACCEPT and the value is discarded. In either case, an MTU element is added to the configuration response, reflecting a valid setting to the other side in case the configuration is rejected.

The above procedure is called “The standard configuration process” of L2CAP connections. In this configuration process the endpoints will respond to a configuration request with a response that either accepts or rejects the offered configuration. If a configuration was rejected, the endpoints will continue to negotiate until they reach an agreed upon configuration.

However another type of configuration process exists - the lockstep configuration process. This process is required to facilitate the *Extended Flow Specification* (**EFS**) feature of L2CAP, which allows devices to establish a more comprehensive connection. The EFS feature parameters will need to be validated with each of the endpoints local Bluetooth controller, and so the endpoints response to a configuration request may be “Pending”. Once both EFS parameters have been exchanged between the endpoints, and the validation of EFS is achieved, a final response will be returned by each of the endpoints.

## Linux kernel RCE vulnerability - CVE-2017-1000251

*BlueZ vulnerability - configuration response parsing*

The vulnerability lies in BlueZ's implementation of L2CAP's EFS feature, in l2cap_parse_conf_rsp, which was introduced in kernel version v3.3-rc1, and thus affects all version from there on. l2cap_parse_conf_rsp can be seen here in abbreviated form:

```
static int l2cap_parse_conf_rsp(struct l2cap_chan *chan, void *rsp, int len,
                void *data, u16 *result)
{
    struct l2cap_conf_req *req = data;
    void *ptr = req->data;
    // ...
    while (len >= L2CAP_CONF_OPT_SIZE) {
        len -= l2cap_get_conf_opt(&rsp, &type, &olen, &val);

        switch (type) {
        case L2CAP_CONF_MTU:
            // Validate MTU...
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_MTU, 2, chan->imtu);
            break;

        case L2CAP_CONF_FLUSH_TO:
            chan->flush_to = val;
            l2cap_add_conf_opt(&ptr, L2CAP_CONF_FLUSH_TO,
                       2, chan->flush_to);
            break;

        // ...
        }
    }
    // ...
    return ptr - data;
}
```

Excerpt from *l2cap_parse_conf_rsp* (net/bluetooth/l2cap_core.c)

This function receives a configuration response buffer in the rsp argument, and its length in the len argument. It extracts elements from the buffer one by one using the l2cap_get_conf_opt function, until the len argument runs out. Each element it unpacks from the configuration response is validated and then packed back onto a response buffer, which is pointed to by the data argument.

*However, the size of this response buffer is not passed into the function.*

Essentially, all elements in the rsp would be copied onto the data buffer via &ptr (offset to l2cap_conf_req.data) regardless of the target's buffer size.

Note that the size of the incoming response is not limited - elements can be duplicated, which allows an attacker to control the size of the rsp buffer, and as a result the amount of data copied

onto data. The origin of the data buffer - l2cap_parse_conf_rsp is called from two locations, both in a function called l2cap_config_rsp, which, as its name implies, handles configuration response messages. Both invocations are similar, so both can be used to exploit this vulnerability

```
switch (result) {
case L2CAP_CONF_SUCCESS:
    ...
    break;

case L2CAP_CONF_PENDING:
    set_bit(CONF_REM_CONF_PEND, &chan->conf_state);
    if (test_bit(CONF_LOC_CONF_PEND, &chan->conf_state)) {
        char buf[64];
        len = l2cap_parse_conf_rsp(chan, rsp->data, len,
                    buf, &result);
    ...
    goto done;
```

Excerpt from *l2cap_config_rsp* (net/bluetooth/l2cap_core.c)

The switch examines the result value, which was previously unpacked from the configuration response packet, and can thus be controlled by an attacker. The response buffer is a small stack buffer, named buf, declared in the scope of the if statement which leads to the call.

The configuration for the current channel is then tested for the “Pending” state (as described above in the lockstep configuration process). So to access this flow, an attacker needs his target to be in the “Pending” state, which he can do by triggering the following code path:

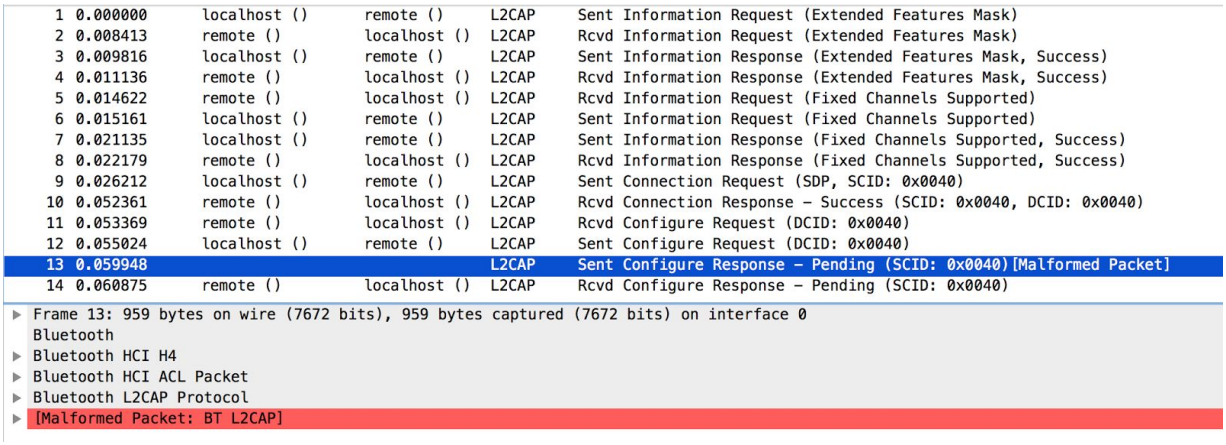
```
if (remote_efs) {
    if (chan->local_stype != L2CAP_SERV_NOTRAFIC &&
        efs.stype != L2CAP_SERV_NOTRAFIC &&
        efs.stype != chan->local_stype) {
        ... // We don't want this branch, easy to avoid
    } else {
        /* Send PENDING Conf Rsp */
        result = L2CAP_CONF_PENDING;
        set_bit(CONF_LOC_CONF_PEND, &chan->conf_state);
    }
}
```

Excerpt from *l2cap_parse_conf_req* (net/bluetooth/l2cap_core.c)

This action is simple - an attacker only needs to send a configuration request with an EFS element, setting the stype field to L2CAP_SERV_NOTRAFIC.

After the “Pending” state is achieved, the next configuration response sent with the result field set to L2CAP_CONF_PENDING will trigger the vulnerability, leading buf[64] to be overwritten with an arbitrarily sized buffer.

This vulnerability allows an attacker to overflow a 64 byte buffer on the kernel stack by an unlimited amount of data, so long as it conforms to the structure of a valid L2CAP configuration response.




Capture of the attack process - note the malformed “Pending” configuration response

## Exploitability

Today, a stack overflow like the vulnerability described above doesn’t automatically translate into code execution. Modern Operating Systems have mitigation techniques specifically to prevent memory corruption vulnerabilities resulting in code execution. Despite this, the Linux Kernel is lagging behind in implementing some modern mitigations in its default configuration. Both stack canaries - which protect against stack overflows, and KASLR (kernel address space layout randomization) are lacking in most devices running Linux today. This makes the stack overflow presented above easy to exploit - as we demonstrate in the demo video.

It should be mentioned that testing and triggering this vulnerability was not an easy task, and required  direct use of the ACL layer to send malformed L2CAP packets. Since no Bluetooth stack provides this to the user a minimal stack implementing the HCI, ACL and L2CAP layers had to be created. The high barrier of entry for testing highly exposed kernel code paths is also detrimental to security. We will be releasing the testing framework we developed, alongside an exploit code of this specific vulnerability in a future blog post. This testing framework could assist researchers in further exploration and pentesting of Bluetooth stacks.

Watch a video of the Linux exploit here.

## Impact

In BlueZ’s case, L2CAP is included as part of the core Linux kernel code. This is a rather dangerous choice. Combining a fully exposed communication protocol, arcane features like EFS and a kernel space implementation is a recipe for trouble. This vulnerability is a classic stack

overflow occurring in the context of a kernel thread. This provides an attacker with a full and reliable kernel-level exploit for any Bluetooth enabled device running Linux, requiring no additional steps. Moreover, each compromised host can be used to launch secondary attacks, making this vulnerability wormable.

## SDP

### Overview

SDP (Service Discovery Protocol) is a core layer in Bluetooth, that is a part of every stack. Its purpose is to allow devices to discover the various services and applications that a Bluetooth device supports. In addition, SDP is responsible to translate the fixed UUIDs (Universal Unique Identifiers) of the Bluetooth services, to PSMs (Protocol Service Multiplexer - Bluetooth's equivalent of a L2CAP port number) that can be a dynamically selected number. The retrieved PSM is then used to create an L2CAP connection to the discovered service.

To discover services, an SDP client sends an SDP request, and an appropriate response is returned. SDP defines yet another fragmentation mechanism for the SDP responses returned by an SDP server, called "SDP Continuation".

The SDP continuation works differently than normal fragmentation:

1. First an SDP client will send an SDP request;
2. If a response to this request exceeds the MTU of the established L2CAP connection, a fragment of the response will be returned, and a "continuation state" structure will be prepended to the SDP response.
3. To receive the remaining fragments, the SDP client will send **the same** request again, appending to it the "continuation state" he received in the last response (this type of request is called a continuation request).
4. The SDP server would then return the next fragment of the response.
5. This flow would be repeated until all fragments are delivered.

It is unclear why Bluetooth required another fragmentation layer, as two additional fragmentation layers are defined below SDP - implemented in L2CAP (that calls it "segmentation"), and in the ACL layer. Moreover, the specification leaves one important detail in the SDP continuation mechanism up to the implementers - the specific structure of the continuation state. The specification describes this driely:

> *"The format of the continuation information is not standardized among SDP servers. Each continuation state parameter is meaningful only to the SDP server that generated it."*
>
> Bluetooth Specification v5.0, Vol 3, Part B, page 1926

This decision in the specification of SDP is quite odd since the returned continuation state is not used by the SDP client directly, and its purpose is to be used internally by the server, upon processing of continuation requests. This can lead to the abuse of the continuation state, since the client is left to return it unchanged on each continuation request. Two similar abuses of this

nature led to two information leak vulnerabilities discovered in the Bluetooth stacks of both Linux and Android

### Linux Bluetooth stack (BlueZ) information Leak vulnerability - CVE-2017-1000250

This vulnerability is a direct result of the scenario described above - and is a very common mistake in implementations of fragmentation mechanisms. Since the SDP continuation struct is defined by the implementation, BlueZ decided to define this structure as its continuation state:

```
typedef struct {
	uint32_t timestamp;
	union {
		uint16_t maxBytesSent;
		uint16_t lastIndexSent;
	} cStateValue;
} sdp_cont_state_t;
```

SDP Continuation Struct, as defined in BlueZ (src/sdpd-request.c)

This structure is comprised of a timestamp, which conveniently leaks the machine's timestamp, and an index representing the total number of bytes that were sent so far.

Since an attacker can control the continuation state sent on each request, he can change the index in the continuation struct, and cause the SDP server to return an out of bounds read from the response buffer:

```
...
	} else {
		/* continuation State exists -> get from cache */
		sdp_buf_t *pCache = sdp_get_cached_rsp(cstate);
		if (pCache) {
			uint16_t sent = MIN(max, pCache->data_size -
							cstate->cStateValue.maxBytesSent);
			pResponse = pCache->data;
			memcpy(buf->data,
				pResponse + cstate->cStateValue.maxBytesSent,
				sent);
			buf->data_size += sent;
			cstate->cStateValue.maxBytesSent += sent;
			if (cstate->cStateValue.maxBytesSent == pCache->data_size)
				cstate_size = sdp_set_cstate_pdu(buf, NULL);
			else
				cstate_size = sdp_set_cstate_pdu(buf, cstate);
		} else {
			status = SDP_INVALID_CSTATE;
			SDPDBG("Non-null continuation state, but null cache buffer");
		}
	}
...
```

Excerpt from SDP Search Attribute Request handler - service_search_attr_req (src/sdpd-request.c)

This code from the Search Attribute Request handler of BlueZ SDP Server fails to validate *maxBytesSent* in *cstate* (the continuation state), and allows the above *memcpy* to copy data beyond the allocated size of *pResponse*. The only thing an attacker needs to do to achieve this information leak, is to avoid the *if* that validates all data has been sent (*maxBytesSent* ==

*data_size*) - and this can be easily done since he controls *maxBytesSent*. Since *pResponse* is allocated in the heap this information leak can lead to disclosure of highly sensitive data.

BlueZ is comprised of two parts - one running in the kernel (as has been seen in the L2CAP vulnerability), and the other in the userspace. The *bluetoothd* process contains all of BlueZ's user parts (essentially all layers of the stack above L2CAP). This process holds critical data that can be leaked using this vulnerability, such as encryption keys used in Bluetooth communications, enabling an attack that very much resembles heartbleed.

### Android information Leak vulnerability - CVE-2017-0785

Android's SDP server defines a similar continuation state structure:

```
typedef struct {
    uint16_t cont_offset;
} sdp_cont_state_t;
```

SDP Continuation Struct used in Android's Bluetooth stack

In this case, only a continuation offset (that has similar meaning to the index used in BlueZ) is sent in the continuation struct. Although the code of Android's SDP server search request handler does perform some validations on *cont_offset*, an information leak is still achievable. In the following excerpt, *num_rsp_handles* will hold the total number of handles (that are the sdp records) of the SDP response:

```
  /* Check if this is a continuation request */
  if (*p_req) {
    ...
    if (cont_offset != p_ccb->cont_offset) {
      sdpu_build_n_send_error(p_ccb, trans_num, SDP_INVALID_CONT_STATE,
                              SDP_TEXT_BAD_CONT_INX);
      return;
    }

    rem_handles =
        num_rsp_handles - cont_offset; /* extract the remaining handles */
  } ...
  /* Calculate how many handles will fit in one PDU */
  cur_handles =
      (uint16_t)((p_ccb->rem_mtu_size - SDP_MAX_SERVICE_RSPHDR_LEN) / 4);

  if (rem_handles <= cur_handles)
    cur_handles = rem_handles;
  else /* Continuation is set */
  {
    p_ccb->cont_offset += cur_handles;
    is_cont = true;
  }
...
  for (xx = cont_offset; xx < cont_offset + cur_handles; xx++)
    UINT32_TO_BE_STREAM(p_rsp, rsp_handles[xx]);
```

Excerpt from SDP Search Request handler - process_service_search (stack/sdp/sdp_server.c)

The code holds a copy of the *cont_offset* in its connection object (*p_ccb*), and validates that the received *cont_offset* is equal to the current state of the connection. So a simple abuse of *cont_offset* is not achievable (as done in BlueZ). However, since each continuation request is essentially a new request which only has a continuation state appended to it, the code can be led to a state confusion by changing the parameters of the request between two consecutive requests.
The *num_rsp_handles* is calculated each time a request is received, based on the total size of the specific response. The response's size may vary based on the requested service search that is being performed, and unlike *cont_offset*, *num_rsp_handles* is not saved in the connection object and validated to remain the same throughout the reading of the response fragments. As a result of this state confusion, **an underflow** of *rem_handles* can be achieved:

```
rem_handles = num_rsp_handles - cont_offset;
```

The code assumes that *num_rsp_handles*, and *cont_offset* both refer to the same response that is being sent in fragments. Due to the induced state confusion, and since *rem_handles* is uint16_t, the code will now assume a very large response is needed (up to 64KB) - and the for-loop that follows will copy out of bounds bytes from *rsp_handles* to an outgoing response packet.

To sum up, this info leak can be triggered by an attacker in this flow:

1. A search request is performed to some service.
2. Due to this request, a response is returned with a continuation state. The size of this response will be defined by the MTU of the connection, as seen in the code excerpt above, so an attacker holds some control over the fragments' size as well.
3. A second request is performed to a different service, and the continuation state from the previous response will be prepended to this request. This second search request will be of  a service that will return a smaller response size than the previous response - and this will lead to the described state confusion.
4. A validation of cont_offset will be attempted, but it will pass successfully (since the same cont_offset was appended to the second request).
5. Due to fact *num_rsp_handles* in this second request is smaller than the one in the first request, **an underflow** of rem_handles will be achieved.
6. The code will now assume a very large response is needed - and the for-loop that follows would copy bytes from rsp_handles to an outgoing response packet.
7. From this point on, an attacker can repeat sending the same request, and prepend the returned cont_offset - continuing to read more and more out of bound bytes from rsp_handles.

Similar to the information leak vulnerability in BlueZ, this vulnerability can lead to the disclosure of a large part of the memory - in this instance from the process stack. This data can potentially

include encryption keys, address space and valuable pointers (of code, or data), that can be used to bypass ASLR while exploiting a separate memory corruption vulnerability.

### Conclusion

Fragmentation is always a soft spot in implementations of protocols. However the faulty design of SDP's fragmentation mechanism makes it a terribly hard mechanism to implement without bugs. Even when specific validations are put in place (as in Android's implementation) - eliminating all bugs that can be a result of convoluted state confusions is almost an impossible mission.

## SMP

### Overview

SMP (Security Management Protocol) enables the various security mechanisms of Bluetooth - authentication, authorization, and bonding (also known as "pairing").

SMP is responsible for the process of pairing two devices, and for the authentication mechanisms used when paired devices are connecting to each other. Bluetooth's security mechanisms have evolved a great deal since it's initial versions - and SMP specifically has gone through a lot of changes. Most of the security flaws in Bluetooth were found in this layer of the protocol, mainly in the PIN code exchange mechanism which existed until version 2.1. This version introduced a new authentication mechanism called SSP (Secure Simple Pairing). SSP was almost a complete do-over of SMP's security mechanisms. One of the confusing alterations of SSP was changing the term "Pairing". Prior to SSP "Pairing" referred to the exchange **and storage** of long term keys between devices to create a long term bond between them. Since SSP was introduced, "Pairing" was split in two: "pairing" refers to key exchange, and "bonding" to storage of keys. This resulted in a new type of pairing - short term pairing which exists only during a single Bluetooth connection, without bonding.

The substantial change introduced in SSP was new key exchange mechanisms. Early versions of Bluetooth used a rather naive mechanism: An exchange of a PIN codes (that could unfortunately be derived if captured over the air). Devices that lacked an interface to insert a PIN code just opted for a default PIN code that was hard coded into the device. SSP uses modern key exchange mechanisms (known as "Diffie-Hellman Key Exchange"), and offers various methods for validating the safe passage of the exchanged keys to validate that the exchange was not intercepted by a third party.

These new mechanisms still do not resolve the problem of devices that lack user interface, and thus can't insert a PIN code or verify a safe passage of one on their display. SSP's authentication mechanism for such devices is "Just Works", the weakest mechanism of the lot. For this reason, using "Just Works" is a last resort in SMP.

Here is how it (just) works:

When performing a Secure Simple Pairing procedure, the two endpoints exchange messages to gather their mutual IO capabilities.

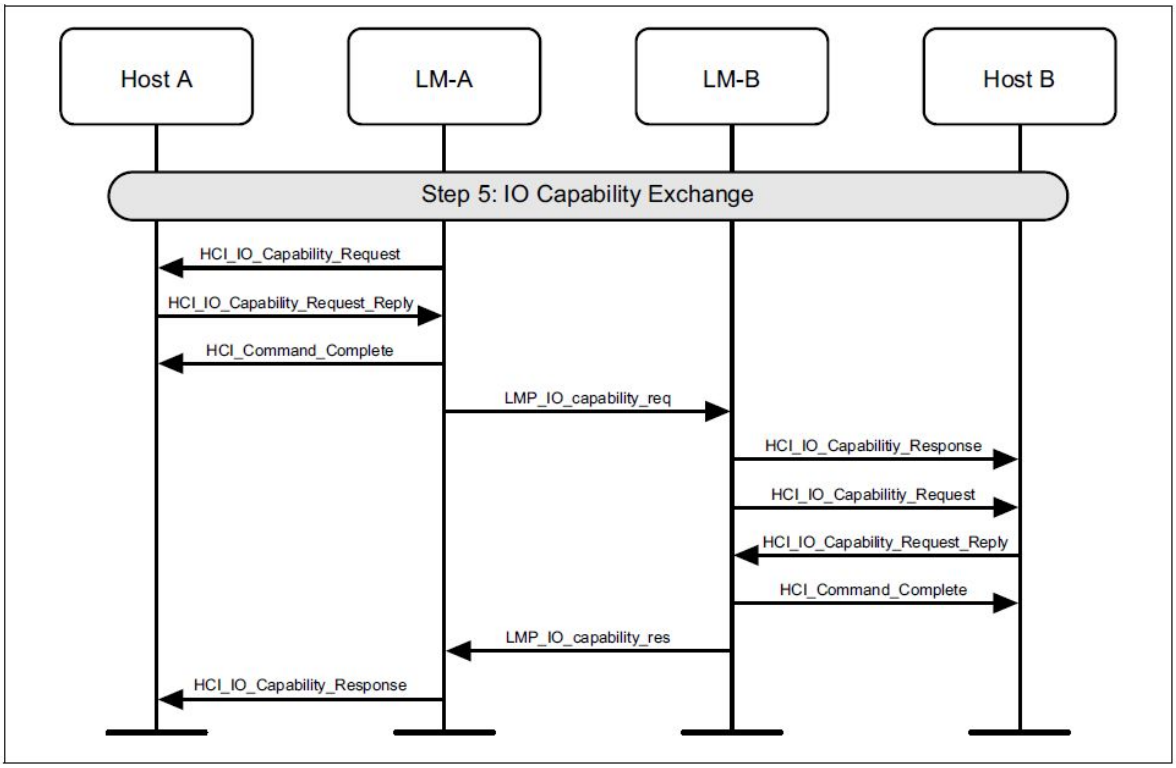



*Figure 4.12: IO capability exchange*

Bluetooth Specification, Version 5.0, Vol 2, Part F, page 1436

Each endpoint will answer these two questions:

- What type of interface it holds
- What type of authentication it requires

The interface options are the following:

*IO_Capability:* *Size: 1 Octet*

| Value | Parameter Description |
|---|---|
| 0x00 | DisplayOnly |
| 0x01 | DisplayYesNo |
| 0x02 | KeyboardOnly |
| 0x03 | NoInputNoOutput |
| 0x04 – 0xFF | Reserved for future use |

And the authentication requirements possibilities are those:

*Authentication_Requirements:* *Size: 1 Octet*

| Value | Parameter Description |
|---|---|
| 0x00 | MITM Protection Not Required – No Bonding. Numeric comparison with automatic accept allowed. |
| 0x01 | MITM Protection Required – No Bonding. Use IO Capabilities to determine authentication procedure |
| 0x02 | MITM Protection Not Required – Dedicated Bonding. Numeric comparison with automatic accept allowed. |
| 0x03 | MITM Protection Required – Dedicated Bonding. Use IO Capabilities to determine authentication procedure |
| 0x04 | MITM Protection Not Required – General Bonding. Numeric Comparison with automatic accept allowed. |
| 0x05 | MITM Protection Required – General Bonding. Use IO capabilities to determine authentication procedure. |
| All other values | Reserved for future use |

The “authentication requirement” that the endpoints exchange determines if they are creating a bond (a long term exchange of keys), and whether they require “MITM Protection” (Man in the middle protection - a high security level against interception eavesdropping of communications).

SSP defines in what method the two endpoints,which might have different interfaces, exchange keys:

| | | Device A (Initiator) | | | |
|---|---|---|---|---|---|
| | | **Display Only** | **DisplayYesNo** | **KeyboardOnly** | **NoInputNoOutput** |
| **Device B (Responder)** | **DisplayOnly** | Numeric Comparison with automatic confirmation on both devices.<br><br>Unauthenticated | Numeric Comparison with automatic confirmation on device B only.<br><br>Unauthenticated | Passkey Entry: Responder Display, Initiator Input.<br><br>Authenticated | Numeric Comparison with automatic confirmation on both devices.<br><br>Unauthenticated |
| | **DisplayYesNo** | Numeric Comparison with automatic confirmation on device A only.<br><br>Unauthenticated | Numeric Comparison: Both Display, Both Confirm.<br><br>Authenticated | Passkey Entry: Responder Display, Initiator Input.<br><br>Authenticated | Numeric Comparison with automatic confirmation on device A only and Yes/No confirmation whether to pair on device B. Device B does not show the confirmation value.<br><br>Unauthenticated |
| | **Keyboard Only** | Passkey Entry: Initiator Display, Responder Input.<br><br>Authenticated | Passkey Entry: Initiator Display, Responder Input.<br><br>Authenticated | Passkey Entry: Initiator and Responder Input.<br><br>Authenticated | Numeric Comparison with automatic confirmation on both devices.<br><br>Unauthenticated |

Table 5.7 (partially displayed), Bluetooth Specification v5.0, Vol 3, Part C, page 2016

When one of the endpoints lacks an interface for key exchange (NoInputNoOutput), the chosen key exchange mechanism is “Numeric Comparison with automatic confirmation”, which is also called “Just Works”. The table above also mentions which of these methods results in an “Authenticated” key, and which does not. An “Authenticated” key is one that the user was able to validate safe passage of.

If one of the endpoints requested “MITM Protection”, an “Unauthenticated” key cannot be used, if not - “Just Works” will be chosen for the key exchange.

### It “Just Works” (but sometimes it doesn’t)

As stated above, “Just Works” is a subset of another authentication mechanism in SSP called “Numeric Comparison” that is used when devices are not limited by their IO capabilities. After establishing that “Just Works” will be the authentication mechanism, the endpoints will perform an altered version of the “Numeric Comparison” key exchange procedure. In the ordinary “Numeric Comparison” a shared “secret” PIN code is exchanged between the endpoints using Diffie-Hellman, and the user validates that the same PIN code appears on each of the devices screen. However, in “Just Works” at least one of the devices lacks a user interface, so validating the PIN code is not an option. Instead an “Automatic confirmation” will be performed by the devices. If one of the devices has sufficient IO capabilities (“DisplayYesNo”), it may authorize the pairing - but it will not display the PIN code, as there is no way to validate it’s safe passage.

The Bluetooth specification notes this about Just Works:

> “The Just Works association model uses the Numeric Comparison protocol but the user is never shown a number and the application **may** simply ask the user to accept the connection (**exact implementation is up to the end product manufacturer**). “
>
> Bluetooth Specification, Version 5.0, Vol 1, Part A, page 245

So despite the obvious need to authorize a new pairing, the spec leaves it up to the various implementations to define how and if the user should accept a new paired device (whether it is short-term pairing or a long-term one).

Android's Bluetooth stack for example implements this decision as follows:

```
/* If JustWorks auto-accept */
 if (p_ssp_cfm_req->just_works) {
   // Pairing consent for JustWorks needed if:
   // 1. Incoming (non-temporary) pairing is detected AND
   // 2. local IO capabilities are DisplayYesNo AND
   // 3. remote IO capabilities are DisplayOnly or NoInputNoOutput;
   if (is_incoming && pairing_cb.bond_type != BOND_TYPE_TEMPORARY &&
       ((p_ssp_cfm_req->loc_io_caps == HCI_IO_CAP_DISPLAY_YESNO) &&
        (p_ssp_cfm_req->rmt_io_caps == HCI_IO_CAP_DISPLAY_ONLY ||
         p_ssp_cfm_req->rmt_io_caps == HCI_IO_CAP_NO_IO))) {
     BTIF_TRACE_EVENT(
         "%s: User consent needed for incoming pairing request. loc_io_caps: "
         "%d, rmt_io_caps: %d",
         __func__, p_ssp_cfm_req->loc_io_caps, p_ssp_cfm_req->rmt_io_caps);
   } else {
     BTIF_TRACE_EVENT("%s: Auto-accept JustWorks pairing", __func__);
     btif_dm_ssp_reply(&bd_addr, BT_SSP_VARIANT_CONSENT, true, 0);
     return;
   }
 }
```

Excerpt from btif_dm_ssp_cfm_req_evt in Android's Bluetooth stack (btif\src\btif_dm.c)

The user will need to accept a JustWorks pairing procedure only if these terms are met:

- The pairing is non-temporary (involving a long-term key exchange - "Bonding")
- The local (the Android's) IO capabilities are DisplayYesNo, and the remote IO capabilities are limited (Display Only, or no IO).

So in case of a temporary pairing procedure, Android will auto-accept, and an attacker will be able to elevate his credentials within Android's Bluetooth stack - as he bypassed the authentication process, while his victim is completely unaware. To reiterate, an attacker can force a temporary pairing to a victim device without any user interaction.

In an empirical testing performed against a Windows machine, the same behavior was detected:

| No. | Time | Source | Destination | Length | Protocol | Info |
|---|---|---|---|---|---|---|
| 17 | 0.822494 | host | controller | 6 | HCI_CMD | Sent Authentication Requested |
| 19 | 0.828453 | controller | host | 9 | HCI_EVT | Rcvd Link Key Request |
| 20 | 0.828496 | host | controller | 10 | HCI_CMD | Sent Link Key Request Negative Reply |
| 22 | 0.830456 | controller | host | 9 | HCI_EVT | Rcvd IO Capability Request |
| 23 | 0.830474 | host | controller | 13 | HCI_CMD | Sent IO Capability Request Reply |
| 25 | 0.841548 | controller | host | 12 | HCI_EVT | Rcvd IO Capability Response |
| 26 | 0.978520 | controller | host | 13 | HCI_EVT | Rcvd User Confirmation Request |
| 27 | 0.978637 | host | controller | 10 | HCI_CMD | Sent User Confirmation Request Reply |
| 29 | 1.404705 | controller | host | 10 | HCI_EVT | Rcvd Simple Pairing Complete |
| 30 | 1.440446 | controller | host | 26 | HCI_EVT | Rcvd Link Key Notification |
| 31 | 1.441453 | controller | host | 6 | HCI_EVT | Rcvd Authentication Complete |
| 32 | 1.441483 | host | controller | 7 | HCI_CMD | Sent Set Connection Encryption |
| 34 | 1.484461 | controller | host | 7 | HCI_EVT | Rcvd Encryption Change |
| 35 | 1.484519 | host | controller | 6 | HCI_CMD | Sent Read Encryption Key Size |
| 43 | 1.500984 | localhost () | LiteonTe_78:… | 17 | L2CAP | Sent Connection Request (BNEP, SCID: 0x0040) |
| 45 | 1.506067 | LiteonTe_78:0… | localhost () | 21 | L2CAP | Rcvd Connection Response - Pending (SCID: 0x0040) |
| 46 | 1.507364 | LiteonTe_78:0… | localhost () | 21 | L2CAP | Rcvd Connection Response - Success (SCID: 0x0040, DCID:… |

```
Bluetooth HCI Command - IO Capability Request Reply
  Command Opcode: IO Capability Request Reply (0x042b)
  Parameter Total Length: 9
  BD_ADDR: LiteonTe_78:0d:d8 (c8:ff:28:78:0d:d8)
  IO Capability: No Input, No Output (3)
  OOB Data Present: OOB Authentication Data Not Present (0)
  Authentication Requirements: MITM Protection Not Required - No Bonding. Numeric Comparison, Automatic Accept Allowed (0)
```

Wireshark capture of “Just Works” procedure with “Auto-Confirm”, against a Windows 10 machine

As seen in the Wireshark capture above, an attacker is able to connect and **authenticate** his connection to a Windows machine via Bluetooth. The attacker chooses to reply to the IO Capability request with: “NoInputNoOutput, MITM Protection Not Required - No Bonding”, and thus the “Just Works” authentication mechanism is chosen. Since the Windows machine automatically confirms the procedure, an Authentication Complete message is almost instantly delivered back to the attacker’s machine.

Now that the attacker is authenticated (even if only with a short-term key), he can access some of the high-level services each machine implements. In the example above, this short-term authentication allows the attacker to access the BNEP service (detailed in the next section). We also found that many other services will allow an L2CAP connection to reach the “Connected” state, once the authentication is achieved through “Just Works”. This is true both for Android and Windows, and perhaps other unexamined stacks that might behave in the same manner.

In many cases, the exposed services will eventually reject access to higher-level features that are meant to be accessed via a fully paired device (one that has performed “bonding”). However, these validations depend on the individual services - and not on the common underlying layer of SMP. This widens the potential attack surface, as more code flows can be reached in each of the many services implemented in each stack.

## Conclusion

It is clear that the “Just Works” mechanism has it merits, as it allows the key exchange through a safe and modern procedure (Diffie-Hellman). It is also obvious that without any IO, a device

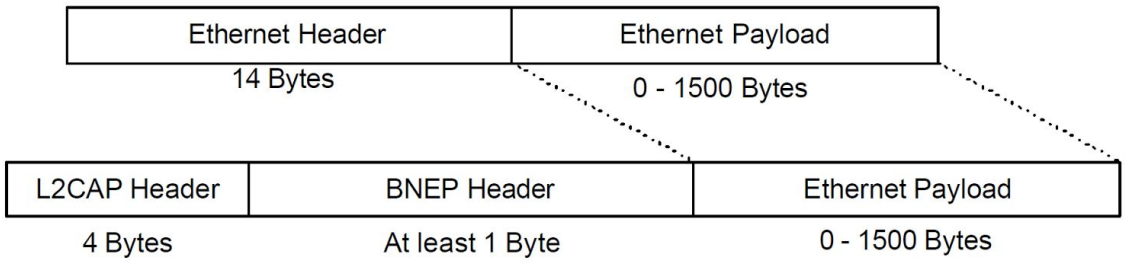
cannot authenticate a PIN code, and thus the mechanism has no MITM protections. Despite this, when one of the parties in a pairing procedure does have IO capabilities, which is the common case, it should be required to confirm the pairing by the end user of this device.
It is unclear if the “auto-confirm” behaviour observed in Android and Windows is intentional, or just a quirk in Bluetooth that these stacks haven’t figured out how to use yet. As we will demonstrate in the next sections, this dark corner of Bluetooth’s SSP led us to discover a significant number of vulnerabilities in the services that are now exposed due to this behaviour in Android and Windows.

## BNEP

### Overview

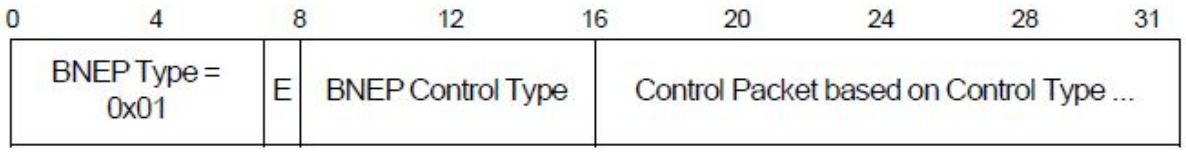
The BNEP (Bluetooth network encapsulation protocol) service facilitates network encapsulation (usually IP based) over Bluetooth. In most cases, this is used to allow internet tethering (sharing) over Bluetooth.

Above BNEP lays the PAN profile that implements the network layer, and the various roles that exist in an IP based network created over Bluetooth. The purpose of the BNEP service in this hierarchy is mostly to encapsulate various forms of Ethernet packets over an L2CAP connection. For this purpose various messages are defined in BNEP for encapsulating compressed and uncompressed Ethernet headers.



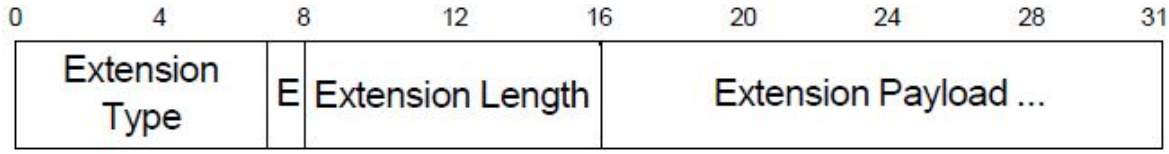

BNEP Specification, Version 1.0, page 13

The above figure demonstrates how the BNEP header is translated into the Ethernet header, based on the specific type of BNEP message used. So basically, BNEP is a simplified, and abbreviated form of Ethernet that is just transmitted over Bluetooth.
Other than the various encapsulation messages, BNEP also supports the BNEP control message. The control message facilitates the creation of a PAN connection (the network layer that lives on top of BNEP) and various flow control features.




BNEP control message format, BNEP Specification, Version 1.0, page 17

To enable multiple control messages in a single L2CAP message, an optional extension header may also be appended to the BNEP header. Each "extension bit" (The “E” in the above figure) turned on in the BNEP header marks the start of an extension header which will include an additional control message.




BNEP extension header format, BNEP Specification, Version 1.0, page 39

In Android's stack, two RCE vulnerabilities were found in the code flow that handles incoming BNEP control messages.

### Android RCE vulnerability #1 - CVE-2017-0781

**The first vulnerability** lies in the following call to memcpy:

```
UINT8 *p = (UINT8 *)(p_buf + 1) + p_buf->offset;
...
type = *p++;
extension_present = type >> 7;
type &= 0x7f;
...
switch (type)
{
...
case BNEP_FRAME_CONTROL:
    ctrl_type = *p;
    p = bnep_process_control_packet (p_bcb, p, &rem_len, FALSE);
    if (ctrl_type == BNEP_SETUP_CONNECTION_REQUEST_MSG &&
        p_bcb->con_state != BNEP_STATE_CONNECTED &&
        extension_present && p && rem_len)
    {
        p_bcb->p_pending_data = (BT_HDR *)osi_malloc(rem_len);
        memcpy((UINT8 *)(p_bcb->p_pending_data + 1), p, rem_len);
        ...
    }
...
```

Excerpt from Android’s BNEP message handler: *bnep_data_ind*

The above code flow is the process of handling incoming BNEP control messages. The BNEP_FRAME_CONTROL is the switch case for BNEP control messages. This specific flow is an attempt to handle a unique use case: since multiple control messages may pass in a single L2CAP message (using the extension bit), the state of the BNEP connection may change between one control message to the other. If for example a SETUP_CONNECTION_REQUEST is sent as the control message, any following control messages might expect to be parsed while the code is in CONNECTED state (and not its initial state which is IDLE). Switching to the CONNECTED state requires the a successful completion of the authentication process (as described in the previous section), and since this process is asynchronous, the state in this context will still be IDLE. The solution for this problem is to parse the remaining control messages at a later time - once the authentication process is complete, and the state of connection has transitioned to CONNECTED.

For this purpose, the above code saves the remaining unparsed message for later use (in *p_pending_data*). However, a simple mistake lies in this code:
First the *p_pending_data* buffer is allocated on the heap, with size *rem_len*. Later, a memcpy is made to *p_pending_data* **+ 1** with the size *rem_len*. Thus the memcpy will overflow the buffer by *sizeof(p_pending_data)* bytes! One may wonder how such a mistake can go unnoticed, as it causes a heap corruption **every** time this code is triggered. Additionally, this causes an inherent memory leak since the previous *p_pending_data* pointer is never freed before another allocation occurs. It is very likely that this code did never actually run, not during real world usage, and probably not even during coverage testing.

The field *p_pending_data* is of type *BT_HDR*, which is 8 bytes long. Additionally, *rem_len*, which controls the size of the allocation, is under the attacker's control, since it's the length of the remaining un-parsed bytes in the packet, as well as the source for the memcpy (*p*) which points to the attacker-controlled packet.

The overflow can be triggered by sending this specially crafted packet in a BNEP connection:

| type | ctrl_type | len | Overflow payload (8 bytes) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 81 | 01 | 00 | 41 | 41 | 41 | 41 | 41 | 41 | 41 | 41 |

Figure 3

The *type* field consists of the *extension_present* bit (which is set), and the *BNEP_FRAME_CONTROL* type (01). The *ctrl_type* field is set to *BNEP_SETUP_CONNECTION_REQUEST_MSG* (01). This allows the flow to reach the vulnerable memcpy call. It should also be noted that *con_state* is indeed not set to *BNEP_STATE_CONNECTED* by default. Inside *bnep_process_control_packet*, the 0 sized *len* passes all the checks, resulting in *rem_len* being decremented properly. As such, the memcpy overflows the heap with the overflow payload bytes.

Notably, since it's possible to send an arbitrarily sized packet, the *osi_malloc* allocation size can be controlled, since *rem_len* represents the size of the *payload* in the packet. This allows an overflow of 8 bytes on the heap following a buffer of **any** chosen size, which makes exploitation much easier.

### Android RCE vulnerability #2 - CVE-2017-0782

**The second vulnerability** also appears in a flow that occurs under *bnep_data_ind*. This one lies in the following integer underflow of *rem_len* in the function *bnep_process_control_packet*:

```
...
if (is_ext)
{
    ext_len = *p++;
    *rem_len = *rem_len - 1;
}
...
control_type = *p++;
*rem_len = *rem_len - 1;
...
switch (control_type)
{
...
default :
    ...
    if (is_ext)
    {
        p += (ext_len - 1);
        *rem_len -= (ext_len - 1);
    }
    break;
}
...
```

Excerpt from Android's processing of BNEP control packets: *bnep_process_control_packet*

This function handles the processing of all BNEP control messages, and the extension header to parse additional sub-messages passed inside a parent control message. The BNEP specification allows unrecognized extension messages to be ignored by the receiving side, and thus the 'default' case above tries to skip unrecognized control messages using the extension length from the extension header.

The integer *rem_len* is defined as a 16-bit unsigned short and represents the actual amount of remaining unparsed bytes in an attacker-controlled packet. The value of *ext_len* is 8 bits unsigned, and is part of the extension header that is also attacker-controlled. Thus a proper *rem_len* can suddenly be underflowed to almost any value above 0xff00, making any further handling of the packet that depends on *rem_len* unsafe.

For example, if *rem_len* originally equals 10, and the attacker sets *ext_len* to be 12, the resulting value will become:

```
rem_len -= (12 - 1) ⇔ rem_len -= 11 ⇔ rem_len == 10 - 11 == 0xffff
```

In the *bnep_data_ind* code, after the call to *bnep_process_control_packet*, the (now unsafe) *rem_len* is indeed used in a dangerous way:

```
...
while (extension_present && p && rem_len)
{
    ext_type = *p;
    extension_present = ext_type >> 7;
    ext_type &= 0x7F;
    /* if unknown extension present stop processing */
    if (ext_type)
    {
         ...
         break;
    }
    p++;
    rem_len--;
    p = bnep_process_control_packet (p_bcb, p, &rem_len, TRUE);
}

p_buf->offset += p_buf->len - rem_len;
p_buf->len = rem_len;
...
else if (bnep_cb.p_data_ind_cb)
{
    (*bnep_cb.p_data_ind_cb)(p_bcb->handle, p_src_addr, p_dst_addr,
protocol, p, rem_len, fw_ext_present);
osi_free(p_buf);
}
```

Excerpt from Android's BNEP message handler: *bnep_data_ind*

The resulting underflowed *rem_len* is then directly set to the *len* of the *p_buf* (the actual packet structure). Additionally, the *offset* field of *p_buf* is affected. This is the offset of the first not-yet parsed byte in the packet. Together, these fields define the amount of bytes left in the packet for upper layers to handle. Following the values from our example above, if the original *len* was 15 (for example), the resulting *offset* will be affected as such:

```
p_buf->offset += (15 - 0xffff) ⇔ p_buf->offset += 16
```

Since now the *offset* is small, and the *len* is large, any upper layer code that handles this packet is forced to deal with an exceptionally large payload. At this point, an attacker can bypass most, if

not all, of the MTU restrictions of packet size. From hereon, the upper layers only assume that the remaining payload is reasonably sized.

Immediately after, a call to *bnep_cb.p_data_ind_cb* (an upper layer handling callback) occurs with the malformed *p_buf* as input. It's thus possible to reach the following code path using the crafted packet:

```
static void bta_pan_data_buf_ind_cback(
    uint16_t handle, const RawAddress& src, const RawAddress& dst,
    uint16_t protocol, BT_HDR* p_buf, bool ext, bool forward)
{
...
    BT_HDR* p_new_buf;
    if (sizeof(tBTA_PAN_DATA_PARAMS) > p_buf->offset) {
    /* offset smaller than data structure in front of actual data */
    p_new_buf = (BT_HDR*)osi_malloc(PAN_BUF_SIZE);
    memcpy((uint8_t*)(p_new_buf + 1) + sizeof(tBTA_PAN_DATA_PARAMS),
           (uint8_t*)(p_buf + 1) + p_buf->offset, p_buf->len);
    ...
    osi_free(p_buf);
...
```

Excerpt from Android's PAN message handler: *bta_pan_data_buf_ind_cback*

As expected, there are no good checks on *offset* and *len* at this point. The only check here verifies that *offset* is smaller than *sizeof(tBTA_PAN_DATA_PARAMS)* (that is 24), which is not a problem. The osi_malloc, however, allocates a buffer *p_new_buf* of size *PAN_BUF_SIZE* (which is 4096) and the memcpy copies *p_buf->len* bytes into it, which were caused to become 0xffff earlier. In short, this results in an overflow of 0xf000 bytes on the heap, following a 4096 bytes sized buffer.

The **source** bytes of the overflowing memcpy are **not** under direct control of the attacker, as they exceed the boundaries of the original packet by far. However, since they are copied from the same area on the heap as the original packet, it should be trivial to create a heap-spray (in advance) since the bytes of the received packets are indeed attacker-controlled. As a result, grooming of the heap prior to the overflow can allow this vulnerability to cause remote code execution.

To create the necessary conditions for reaching the vulnerable flow, the BNEP connection needs to be in the *BNEP_STATE_CONNECTED* state. Therefore, first a valid *BNEP_SETUP_CONNECTION_REQUEST_MSG* needs to be sent. Once this state is reached, the vulnerability can be triggered with a packet such as the following example (6 bytes):

| type | protocol | | ext_type | ext_len | control_type |
|---|---|---|---|---|---|
| 82 | 00 | 00 | 00 | 0A | 10 |

The *type* field indicates a *BNEP_FRAME_COMPRESSED_ETHERNET* packet type, and the flag *extension_present* is set. Since this message is marked with the extension bit, the vulnerable *bnep_process_control_packet* function will be called and the controlled *ext_len* will underflow *rem_len* (as explained earlier). The *control_type* field is set to 0x10 to reach the *default* clause. Once the underflowed *rem_len* returns from this function, it would be copied to *pbuf->len*, and affect *pbuf->offset* as well. Finally, the packet will be passed to the *p_data_ind_cb* which will lead to *bta_pan_data_buf_ind_cback* in the current state, performing the overflowing memcpy.

### Exploitability

As described above, both vulnerabilities can lead to heap overflows with data that is attacker-controlled. In the first RCE vulnerability, an attacker can also control the allocation size of the overflowed buffer, that can assist him with reliable heap shaping. With prior grooming of the heap, both vulnerabilities can eventually lead to code control. An exploit of these vulnerabilities can then execute a ROP chain that would enable an attacker to run any code he'd like in the context of the Bluetooth stack.

The Bluetooth service in Android runs under Zygote (Android service manager), and is surprisingly a 32-bit process (even when the OS and CPU are ARM-64 for instance). This makes exploitation far easier as it limits the ASRL entropy significantly, and in some cases makes it completely inert. More importantly, the service is immediately and automatically restarted by Zygote once it crashes! This provides an attacker with infinite attack attempts, where the reliability of the exploit only affects the time required for a successful run.

When combining the SDP information disclosure vulnerability (CVE-2017-0785) with one the above vulnerabilities, a complete bypass of the ASLR mitigation can be achieved as well. Pointers that are leaked from the stack can be used to allow an attacker to learn the base addresses of the various sections of the Bluetooth process, and these can be used by an attacker to elevate one of the heap overflow vulnerabilities to reliable code control.

We demonstrate these exploits in this video.

### Impact

Successful exploitation results in remote code execution, under the privileges of the com.android.bluetooth service. This service is exceptionally privileged on Android devices: It has access to the filesystem (accessing the user's phonebook, documents, photos, etc.), it has full control of the network stack (that can allow exfiltration of data, MiTM connections and bridging of networks) and it even has the ability to simulate an attached keyboard or mouse that can enable an attacker to gain full control of a device. In addition, since this service has full control of the Bluetooth interface itself, an attacker can also use the victim's Bluetooth interface to attack other devices in its proximity, making this attack vector wormable.

# PAN Profile

## Overview

The next hierarchy in Bluetooth located above the various services, are the Bluetooth profiles. The profiles are another level of abstraction in Bluetooth. For example - the PAN (Personal Area Network) profile defines how a Bluetooth stack uses the BNEP service to create Bluetooth based IP networks.

A Personal Area Network (PAN) is comprised of various roles for each of its connected members:

```
- PAN user (PANU):
  Client of a NAP or client-type member of a GN (see below)
                                  ....
- Network Access Point (NAP):
  Acts as proxy, router or bridge between an existing network infrastructure
  (typically LAN) and (up to 7 active) wireless clients (PANUs).

                                +====================+
                                | LAN Infrastructure |
                                +====================+
                                           |
                                           |
                                     +----------+
                                     |   NAP    |
                                     +----------+
                                    /     |     \
                                   /      |      \
                                  /       |       \
                          +------+    +------+    +------+
                          | PANU |    | PANU |    | PANU |
                          +------+    +------+    +------+
```

From the BlueZ documentation here.

In the general case of two devices in a Bluetooth tethering scenario, one should be the NAP (access point, router) and the other is the PANU (the client).

## The Bluetooth Pineapple - Logical Flaw CVE-2017-0783 & CVE-2017-8628

As explained in the earlier section about SMP, an attacker can bypass authentication, and perform short term pairing with an Android or a Windows device. This will allow him to obtain some access to higher level services and profiles, and the PAN Profile is among these exposed profiles in both Android and Windows stacks. Not all services in these operating systems are exposed in the same manner, due to the fact that each service defines what “Security Level” it requires for incoming and outgoing connections - and not all “Security Levels” allow the use of “Just Works” as the underlying authentication mechanism.

Going a little deeper, it's possible to find the culprit for the rather low "Security Level" requirement of the PAN Profile in Android's Bluetooth stack:

```
#define PAN_SECURITY (BTM_SEC_IN_AUTHENTICATE | BTM_SEC_OUT_AUTHENTICATE |
                      BTM_SEC_IN_ENCRYPT | BTM_SEC_OUT_ENCRYPT)
```

Excerpt from Android Bluetooth stack source code (btif/include/btif_pan_internal.h)

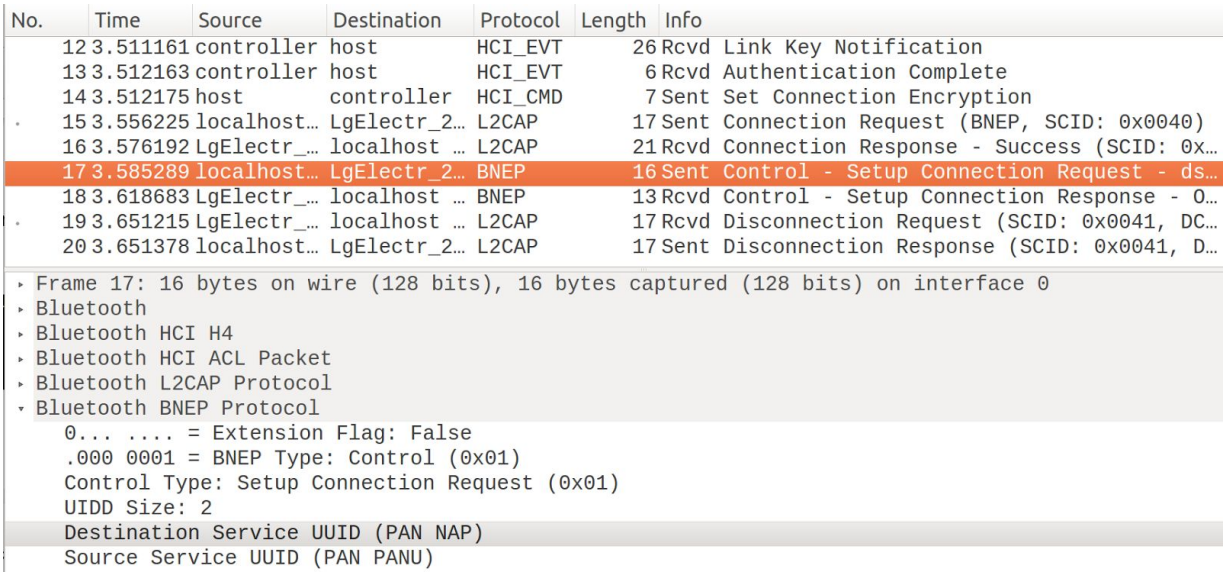
Merely requesting *BTM_SEC_IN_AUTHENTICATE* is just about the minimum security requirement that can be made. Relevant options include:

```
#define BTM_SEC_IN_AUTHENTICATE     0x0002 /* Inbound call requires authentication */
#define BTM_SEC_IN_AUTHORIZE        0x0001 /* Inbound call requires authorization */
#define BTM_SEC_MODE4_LEVEL4        0x0040 /* Secure Connections Only Mode */
#define BTM_SEC_IN_MITM             0x1000 /* inbound Do man in the middle protection */
```

Excerpt from Android Bluetooth stack source code (stack/include/btm_api.h)

Choosing a stronger combination of requirements would have prevented an attacker that has authenticated through "Just Works" the ability to connect to the PAN Profile. Moreover, the *BTM_SEC_IN_AUTHORIZE* would probably demand additional authorization when accessing this service, that would have allowed the victim to reject (via a UI dialog) an attacker's connection. We believe that in the Windows Bluetooth stack, this issue is most likely caused by a similar misconfiguration in the code.

Due to this low "Security Level" requirement, an attacker can leverage the capabilities of the PAN Profile on the targeted device without any authorization. When attempting to connect to the PAN Profile using the obtained short-term key the following occurs:




Wireshark capture of connection attempt to the PAN Profile

Passing the authentication allows the attacker to achieve a successful connection to the BNEP service (open an L2CAP connection) and then the attacker tries to connect to the PAN profile over the BNEP connection (SetupConnectionRequest). Interestingly, the **initiator** of the connection chooses **both** the role of the initiating device, and of the remote device. In this case, a connection where the attacker's side is a PANU and the victim is the NAP was established, but soon after was disconnected, probably because the victim's device detected the tethering feature was turned off. However, since an attacker can chose **both** roles for the PAN participants, additional combinations can be attempted, as seen in this matrix of valid roles:

| Role of the acceptor | Role of the initiator | | | |
|---|---|---|---|---|
| | | NAP | GN | PANU |
| | NAP | NO | NO | YES |
| | GN | NO | NO | YES |
| | PANU | YES | YES | YES |

*Table 1: Valid interactions between the three PAN profile roles*

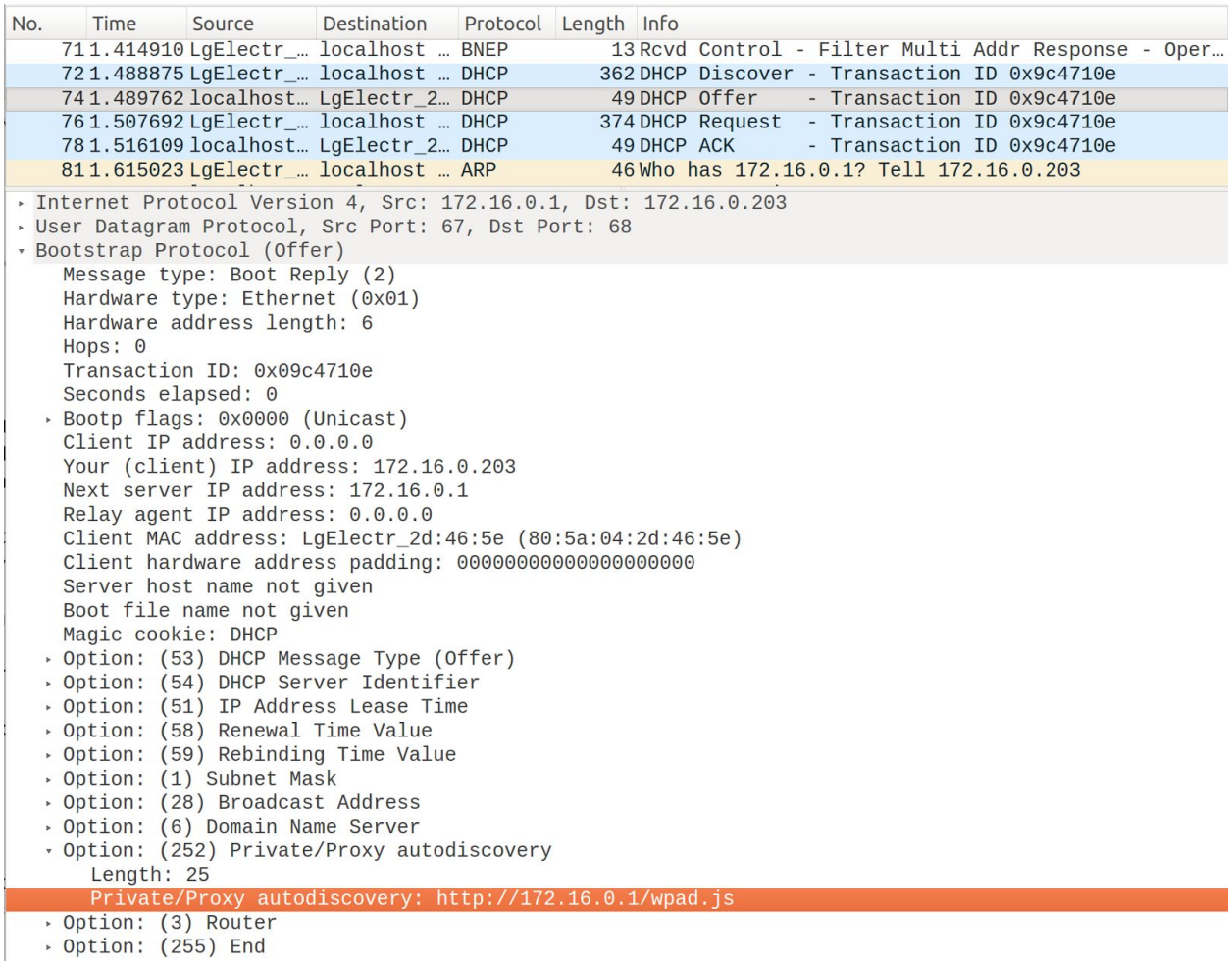
Personal Area Networking Profile v1.0, page 19.

By reversing the roles, and defining the attacker as the NAP and the victim as the PANU the BNEP connection succeeds! This also works when both are set to NAP (even though this combination is marked invalid in the above table). In this case, the victim device is forced to treat the NAP as a new hot-plugged network interface, which results in a DHCP request from the victim:

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 47 | 1.938969 | localhost… | LgElectr_2… | L2CAP | 17 | Sent Connection Request (BNEP, SCID: 0x0040) |
| 57 | 1.962701 | localhost… | LgElectr_2… | BNEP | 16 | Sent Control - Setup Connection Request - dst: <P… |
| 59 | 1.971280 | LgElectr_… | localhost … | BNEP | 13 | Rcvd Control - Setup Connection Response - Operat… |
| 60 | 2.029279 | LgElectr_… | localhost … | ICMPv6 | 114 | Multicast Listener Report Message v2 |
| 61 | 2.068012 | LgElectr_… | localhost … | ICMPv6 | 114 | Multicast Listener Report Message v2 |
| 62 | 2.071490 | LgElectr_… | localhost … | DHCP | 362 | DHCP Discover - Transaction ID 0x36f496db |
| 63 | 2.371544 | LgElectr_… | localhost … | ICMPv6 | 82 | Neighbor Solicitation for fe80::825a:4ff:fe2d:465e |
| 64 | 3.375238 | LgElectr_… | localhost … | ICMPv6 | 74 | Router Solicitation from 80:5a:04:2d:46:5e |
| 65 | 3.378115 | LgElectr_… | localhost … | ICMPv6 | 134 | Multicast Listener Report Message v2 |
| 89 | 7.081480 | LgElectr_… | localhost … | DHCP | 362 | DHCP Discover - Transaction ID 0x36f496db |
| 90 | 7.382734 | LgElectr_… | localhost … | ICMPv6 | 74 | Router Solicitation from 80:5a:04:2d:46:5e |
| 91 | 7.749211 | LgElectr_… | localhost … | ICMPv6 | 94 | Multicast Listener Report Message v2 |
| 92 | 11.3854… | LgElectr_… | localhost … | ICMPv6 | 74 | Router Solicitation from 80:5a:04:2d:46:5e |
| 93 | 12.0926… | LgElectr_… | localhost … | DHCP | 362 | DHCP Discover - Transaction ID 0x36f496db |

At this point, the attacker can set up a DHCP server and push malicious static routes, DNS servers and WPAD. This is essentially equivalent to a WiFi pineapple attack over Bluetooth, only without any user interaction.

A DHCP response packet controlled by an attacker can look like this:




A DHCP client daemon (running on a Windows or Android device) may respect many “options” other than the assigned IP address. Settings like static routes, netmask, default gateway and even the DNS servers are overridden by the **last** DHCP procedure performed by the daemon. Other options like WPAD (a URL to a system-wide HTTP proxy configuration script) are also respected on Windows. These can allow an attacker to open a pop-up browser window with an attacker-controlled page.
Since an attacker can force a DHCP procedure to occur at will, the malicious settings will be the latest ones, and thus the ones used by the victim machine.

Watch a video demonstration of the Windows exploit here.

### Impact

The power of the WiFi Pineapple is well known - it can allow an attacker to be a Man in the Middle on all traffic that is meant to be routed to a specific network, or to the internet - and thus intercept, inject or alter sensitive data that is received by or sent from a targeted device.

However the WiFi Pineapple has crucial limitations: It works by sniffing WiFi probe requests sent by devices to **open** networks, and then masquerading as those networks and responding in their name. So first the WiFi Pineapple needs to detect a probe request, that might not be sent by a device that is already connected to a WiFi network, and even then it will only MiTM **open** networks that have no encryption key. and thus can't be authenticated by the connected device.

The above logical flaw demonstrates the ability to create a Bluetooth Pineapple, that is not subject to those limitations at all. An attacker can force a connection to a targeted device, regardless of its state (other than Bluetooth being turned on). The attack also does not depend on the device being connected to open WiFi networks in the past.

### Conclusion

The specification of the PAN Profile details the PAN's security requirements from the underlying Bluetooth stack layers. However, this document was last updated in 2003 and it's latest version is v1.0. In fact, the "Secure simple pairing" mechanism that is in use by Bluetooth today, and allows short-term authentication through "Just Works", did not even exist back then. The security requirements in the PAN specification have not been updated. This may have contributed to the rather low "Security Level" requirements defined by both Windows and Android stacks for the PAN Profile.

## Proprietary Protocols over Bluetooth

While most vendors rely on the services defined in the Bluetooth specification upon implementing their Bluetooth stacks, certain vendors create their own proprietary protocol layers within the stack - sometimes at its very core. Such is the case of Apple which implemented multiple protocol layers that run alongside Bluetooth's defined connection protocol layer - L2CAP.

### Apple's proprietary protocols over Bluetooth

Although Apple's iOS was not the major focus of our research, we observed a few interesting details when reviewing its Bluetooth stack:

- Unlike Android and Windows, iOS does not allow silent authentication to take place via "Just Works" - once an attacker attempts authentication through "Just Works", the user of the targeted device is informed that a device has initiated pairing with it, and only if the user authorizes the pairing the authentication will succeed. This is of course more close to what the designers of the Bluetooth specification had in mind, and is the logical way to implement "Just Works".
- Moreover, authentication of Bluetooth connections is more tightly coupled in iOS with the creation of L2CAP connections - as it should be. In other stacks we reviewed, the authentication process is something that can be initiated at various times in the life of a Bluetooth connection. This can lead code flows of the stack's services to be exposed to unauthenticated connections, as they allow incoming packets to be parsed in parallel to the completion of the authentication process. In iOS, however, the implementation of SMP is a lot more strict: other than SDP, no L2CAP connection is allowed before the

authentication process is completed successfully. This limits the attack surface to an unauthenticated attacker significantly, making any possible vulnerabilities in the higher layers of the stack unreachable.

Despite taking proper actions in the design and implementation of the security mechanisms of SMP and L2CAP, Apple has also implemented proprietary protocols in the iOS stack that live in parallel to L2CAP, which are not subject to the same security mechanism. So while the hardening of security mechanisms in iOS's Bluetooth stack reduces the exposed attack surface, the various proprietary protocols embedded in it widens it.
As described in the section regarding L2CAP, fixed CIDs are reserved in the protocol for specific purposes. CID number 1 is used as the signaling channel, for example, and some others fixed CIDs are defined as well:

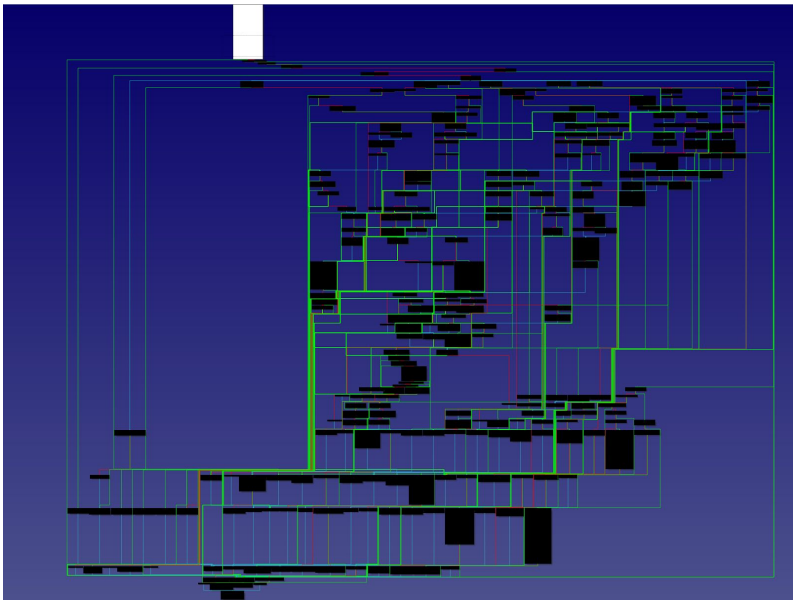
| CID | Description | Channel Characteristics | Logical Link Supported |
|---|---|---|---|
| 0x0003 | AMP Manager Protocol | See [Vol 3] Part E, Section 2.2. | ACL-U |
| 0x0004-0x0006 | Reserved for Future Use | Not applicable | |
| 0x0007 | BR/EDR Security Manager | See [Vol 3] Part H | ACL-U |
| 0x0008-0x003E | Reserved for Future Use | Not applicable | |
| 0x003F | AMP Test Manager | See [Vol 3] Part D, Section 1.2.3 | ACL-U |
| 0x0040-0xFFFF | Dynamically allocated | Communicated using L2CAP configuration mechanism (see Section 7.1) | ACL-U, AMP-U |

*Table 2.1: CID name space on ACL-U, ASB-U, and AMP-U logical links*

L2CAP CID name space, Bluetooth Specification v5.0, Vol. 3, Part A, Section 2.1, page 1728

Apple's proprietary protocols use the range of fixed CIDs that are reserved by the specification for future use (specifically CIDs 0x2A, 0x2B and 0x3A, but possibly others as well). Using fixed CIDs allows Apple to create a completely new hierarchy that replaces L2CAP (in some cases) altogether.

For example, Apple's use of the fixed CID 0x3A (which is called "Piped Dreams" in some of the strings that are referenced in it's code) has substantial code flows that are implemented in it, and resembles L2CAP in many ways:



Graph overview of Apple's implementation of the "Pipe Dreams" protocol (in IDA)

Many of the code flows that are forked from the above function lead to the same handlers that are related with creation of L2CAP connections. So it is possible that state confusions related to the creation of L2CAP connections may be a result of the duplication of code that exists between "Pipe Dreams" and L2CAP. Moreover, it is an entirely new attack surface that is specific to Apple's stack.

Since these protocols are proprietary, they are not documented, and we do not know the full extent of their functionality purposes. However, in one of these protocol a critical remote code execution vulnerability was found.

### Apple's LEAP - RCE in Apple's Low Energy Audio Protocol - CVE-2017-14315

This vulnerability was found in a new protocol Apple invented, which operates on top of Bluetooth, called LEAP (Low energy audio protocol). This protocol is designed to stream audio to low energy audio peripherals, such as low energy headsets, or the Siri Remote for example. Some documentation of this protocol has leaked through Apple's patent filing. It appears that the purpose of this protocol is to enable devices that only have Bluetooth Low Energy to stream audio and send audio commands. However both LEAP and "Pipe Dreams" are still subject to potential attacks by an attacker who connects via Classic Bluetooth connections to a targeted device. Each of these protocols implement some validations that the incoming connections to their fixed CIDs originate from a BLE connections, and not via BR\EDR (a.k.a "Classic") Bluetooth connections. However, these validations are not in the underlying layers of these protocols - but rather in the individual handlers of their various message handlers.

LEAP, for example, allocates two fixed CIDs for it's operation:

- CID 0x2A is reserved for LEAP's signaling channel, and through it LEAP streams can be created to transport LEAP audio data (presumably).

- CID 0x2B is reserved for LEAP's audio data packets that are streamed packets of compressed audio.

Most of LEAP's code lie in the processing of messages sent in its signaling channel. As mentioned above - most of its code validates that incoming messages originate from BLE connections - which limits its attack surface. However, these validations are not consistent across all LEAP code.

In the LEAP handler for incoming audio data (in fixed CID 0x2B) this validation is insufficient. This exposes this handler to attacks, and since it is a fixed CID that is not subject to the security mechanisms of L2CAP or SMP, it is completely unauthenticated.

```
void leap_audio_handler(void *incoming_packet, size_t incoming_packet_length)
{
    …
    void *audio_chunk = new(0x68);
    memcpy(audio_chunk, incoming_packet, incoming_packet_length);
    …
    audio_handler_callback(audio_chunk, incoming_packet_length);
    …
}
```

Psuedo code of LEAP's audio data handler (based on reverse engineering of iOS v9.3.5)

```
MOV             W0, #0x68 ; unsigned __int64
BL              operator new(ulong) ; Branch with Link
MOV             X23_new_packet, X0 ; Rd = Op2
STP             XZR, XZR, [X23_new_packet,#8] ; Store Pair
MOV             W8, #3  ; Rd = Op2
STR             W8, [X23_new_packet,#0x18] ; Store to Memory
MOV             W8, #1  ; Rd = Op2
STRB            W8, [X23_new_packet,#0x1C] ; Store to Memory
ADR             X8, qword_10031B9E0 ; Load address
NOP                     ; No Operation
STRB            WZR, [X23_new_packet,#0x1D] ; Store to Memory
ADD             X8, X8, #0x10 ; Rd = Op1 + Op2
STR             X8, [X23_new_packet] ; Store to Memory
STR             X21, [X23_new_packet,#0x20] ; Store to Memory
STRH            W19, [X23_new_packet,#0x64] ; Store to Memory
ADD             X0, X23_new_packet, #0x28 ; void *
MOV             X1, X20_packet ; void *
MOV             X2, X19_packet_size ; size_t
BL              _memcpy ; Branch with Link
```

Excerpt of assembly code of the above *leap_audio_handler* (from iOS v9.3.5)

In the *audio_handler_callback* the code validates that this incoming packet was received from an authenticated BLE connection. However by this point, the above *memcpy* could already result in a **heap overflow.** This vulnerability is a very simple mistake: the code assumes that all incoming LEAP audio chunks are limited to maximum 0x68 bytes - when this code is triggered through a Bluetooth classic connection, the limitations of incoming packets are not as low, and can create a significant overflow with data that is completely attacker controlled. Since this overflow can be triggered multiple times, this vulnerability can lead to remote code execution in the context of iOS's Bluetooth stack.

### Impact

Due to the fact this vulnerability was mitigated in iOS version 10, a full exploit was not developed by us. Despite this, this vulnerability still poses a great risk to any iOS device prior to version 10, as it is does not require any user interaction or configuration of any sort on the targeted device, and can be leveraged by an attacker to gain remote code execution in a very high privileged context (the Bluetooth process).

## Final Notes

The vulnerabilities described above, and the related exploitation techniques are not very complex. They demonstrate how protocols which are difficult to implement are susceptible to bugs. Implementers of such a complex standard as Bluetooth have to heavily rely on guidelines presented in the specification, which is severely outdated in certain parts, and completely lacking in others. A researcher or attacker armed with domain-specific knowledge of obscure features implemented in Bluetooth can tap into a relatively unexamined attack surface.

It is apparent from our findings that Bluetooth implementations have not received the same level of scrutiny and research other outward-facing protocols have (like WiFi, or TCP/IP stacks). This might be the result of Bluetooth's relative complexity, and the high barrier of entrance for a researcher attempting to research it.  Another contributing factor are two common misconceptions about Bluetooth: One is that connections in Bluetooth have to be of paired devices (which they do not), and the other is that devices MAC address (BDADDR) are safely hidden while they are not in discoverable mode (which they are not).

The result of the lack of proper inspection and testing of the Bluetooth implementations is a major and comprehensive attack vector. While it is becoming harder to gain full control of devices through the main processes, many ignore seemingly peripheral parts of it - such as the Bluetooth stack. Attackers can target these sections of the device, and take control through them, as they are an integral part of the operating system - either as part of the kernel itself, or as highly privileged processes on top of it.. The security community needs to ensure no doors are left open, and treat vulnerabilities such as those described here, which grant attackers a back route to full control.

We hope this paper will be an initial step for a wider and more inclusive audit of the security issues that might lie dormant in the various Bluetooth stacks that are part of the 8.2 Billion Bluetooth devices that are in use today. We encourage other researchers to use this paper as a guideline for the various pitfalls that might exist in implementations of Bluetooth stacks.

# About Armis

Armis, the leading asset visibility and security company, provides the industry's first unified asset intelligence platform designed to address the new extended attack surface that connected assets create. Fortune 100 companies trust our real-time and continuous protection to see with full context all managed, unmanaged assets across IT, cloud, IoT devices, medical devices (IoMT), operational technology (OT), industrial control systems (ICS), and 5G. Armis provides passive cyber asset management, risk management, and automated enforcement. Armis is a privately held company and headquartered in California.

armis.com

info@armis.com