Intel® Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass Side Channel Exploits

White Paper

Revision 1.3

July 2018

THIS PAGE IS LEFT INTENTIONALLY BLANK

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at www.intel.com.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Intel disclaims all implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others

Contents

§

Revision History

Revision Number	Description	Date
1.0	Initial version with guidance for CVE-2017-5753	February 2018
1.1	sizefunc EDL attribute removed	March 2018
1.2	Remove references to sizefunc.	April 2018
1.3	Guidance for CVE-2018-3693	July 2018

THIS PAGE IS LEFT INTENTIONALLY BLANK

1 Glossary/Acronyms

Term/Acronym	Meaning
ECALL	Call into an enclave.
OCALL	Call outside an enclave.
tRTS	Intel® SGX trusted runtime system. A static library included in the Intel® SGX SDK and built into any enclave built with the SDK.
EDL	Enclave Definition Language. A language used to define the interface between applications and enclaves.
Edger8r	SGX SDK Tool used to compile EDL files.

2 Background

On January 3, 2018, a team of security researchers at Google Project Zero [GPZ] disclosed three variants of a side channel analysis method that, when used for malicious purposes, have the potential to improperly gather sensitive data from many types of computing devices with many different vendors' processors and system software. These three vulnerabilities are described in three Intel® Security Advisories [SECADV] [SECADVOSS2] [SECADVOSS3], and are also known as 'Spectre' (referring to variants 1 and 2) and 'Meltdown' (referring to variant 3).

Applications that use Intel® Software Guard Extensions (Intel® SGX) are vulnerable to the GPZ variant 1 side channel analysis methods until mitigations[1] described in this document have been implemented. While any attack on an SGX enclave will be specific to the way the enclave handles its data, the actions outlined in this document should be followed to help minimize potential impacts.

Guidance issued by Intel® [ANALYSIS] identified that software, not microcode, would be responsible for mitigating GPZ variant 1 vulnerabilities. GPZ variant 1 is also known as Bounds Check Bypass (BCB) and CVE-2017-5753 [CVE20175753].

Additional research has disclosed related variations of BCB that are commonly referred to as Bounds Check Bypass Store (BCBS), and described in an Intel® Security Advisory [SECADVOSS] and CVE-2018-3693 [CVE20183693]. The vulnerability is similar to BCB and can be mitigated the same way. Exploits of BCBS differ from exploits of BCB in that the presence of BCBS vulnerabilities allows the attacker to speculatively *write* to an address as opposed to the speculative reads (only) enabled by BCB. As with BCB, software, not microcode, is responsible for mitigating BCBS.

In this document we will identify changes that have been made to the Intel® SGX Software Developer's Kit (SDK) as a result of both BCB and BCBS. We also provide clarifying guidance on what Intel® SGX developers should review in cases that cannot be addressed merely by recompiling with the updated SDK.

[1] *The mitigation techniques discussed in this document address the Bounds Check Bypass and Bounds Check Bypass Store methods of side channel analysis originally disclosed by Google Project Zero and by Vladimir Kiriansky and Carl Waldspurger, respectively. Intel® continues to research these issues and cannot guarantee that these mitigation techniques will apply to vulnerabilities or variations resulting from current or future research.*

3 Intel® SGX SDK Changes

The SDK has been updated to mitigate several BCB vulnerabilities. Analysis has concluded that no further SDK updates are required for BCBS. Stopping speculative execution in the SDK code is achieved by inserting LFENCE instructions where speculative execution might possibly lead to a secret-disclosing side channel. The following table lists the SDK changes and describes the corresponding bypass.

SDK change	Bypass	SDK component impacted
Stop speculative execution that could lead to overflow of the ECALL function table used by enclaves built with the SDK.	Outside code (attacker) controls index to this table. Check of this index could be bypassed.	tRTS library
Stop speculative execution that could lead to enclave operating on a secret in enclave memory as though it was not a secret.	In enclaves built with the SDK, buffers corresponding to pointer inputs (without certain EDL attributes) are checked to ensure that they are outside the enclave. These checks could be bypassed.	Edger8r and tRTS library
Stop speculative execution during first enclave call. The first enclave call is part of the enclave loading process and doesn't execute Edger8r-generated code.	The first enclave call passes a pointer that's treated as though it points to outside memory. Check of this can be bypassed.	tRTS library
Stop speculative execution that could lead to overflow of structure used for sealed data.	sgx_sealed_data_t structure includes length field that is used to calculate pointer values. Checks of these values could be bypassed.	tSeal library
Stop speculative execution in trusted key exchange library. Use of this library adds ecalls to enclaves that use it. One of these ecalls, sgx_ra_get_msg3_trusted, has a user_check pointer input.	Check that user_check pointer points to buffer outside enclave could be bypassed. Since user_check input, other SDK changes don't help. See section 4.1.2.	tkey_exchange library
Stop speculative execution in trusted key exchange library. Use of this library adds ecalls to enclaves that use it. The ecalls take a context parameter that is used as a session array index.	Outside code (attacker) controls context input. Check of context to make sure it doesn't overflow session array could be bypassed.	tkey_exchange library

Table 1. List of SDK Changes

4 Intel® SGX Developer Guidance

In order to take advantage of the SDK changes, developers should rebuild their enclaves with the updated SDK, Intel® SGX SDK for Windows* OS Version 1.9.106 or Intel® SGX SDK for Linux OS Version 2.1.102 or later. Developers may choose to increment their enclaves' ISVSVNs consistent with guidance in the Intel® SGX Developer Guide [SGXDEVGUIDE]. So, at some point, provisioning of secrets to an enclave, or otherwise deciding to trust an enclave, should require an ISVSVN whose associated enclave was built with the updated SDK.

Rebuilding with the updated SDK may not be sufficient, as not all BCB and BCBS exploit scenarios can be mitigated automatically. For BCB, all loads from memory where an attacker can control the address should be analyzed. In contrast, BCBS requires analysis of *stores* to memory where an attacker can control the address and data. For more information, see [ANALYZE], the Intel® white paper covering Bounds Check Bypass (BCB) and Bounds Check Bypass Store (BCBS) more generally.

As a result, enclave developers should analyze all enclave inputs that aren't handled by the SDK as described in section 3. The following sections describe different types of enclave inputs and what the developer should do to add mitigations for each type.

4.1 Enclave Inputs

The following types of enclave inputs can lead to exploits:

- Inputs that are interpreted as addresses/pointers
- Inputs that are used to calculate addresses
- Inputs whose contents (recursive) are interpreted as addresses/pointers or are used to calculate addresses

A BCB vulnerability permits an attacker to make the pointer or calculated pointer point to a secret in enclave memory. Then, code in the enclave that should only run when the pointer points to outside memory can execute speculatively, thus acting as a "disclosure gadget".

With BCBS, an attacker tries to find enclave code that can speculatively write to an address that the attacker specifies, with data under the attacker's control. For example, the attacker could speculatively overwrite the return address of a function such that when the function returns, a disclosure gadget would be executed speculatively. The main difference from BCB is that a BCBS attack starts with a speculative write to an enclave address instead of a speculative read of a secret.

Even if the EDL for an enclave doesn't include any such inputs, the enclave will have them. The code generated from the enclave EDL by the Edger8r puts the parameters of each ecall in a structure and passes a pointer to this "marshaling structure" to the enclave. Also, code in the tRTS along with Edger8r-generated code maintains an "ecall table" and converts the developer's ecall into an indirect call through this table. Attacks on these inputs are mitigated in the updated SDK.

The updated SDK also inserts mitigations for enclave inputs that appear in the enclave's EDL that are interpreted as addresses/pointers provided:

- They are declared as pointer types in the EDL.
- They don't use the user_check or sizefunc[2] EDL attributes.
- The pointer-ness of the input isn't hidden via typedef.

[2] EDL "sizefunc" attribute has been removed. See section 4.1.4.

- In this case, the “isptr” EDL attribute can be used in order to have the input still be treated as a pointer.

However, even if a pointer input meets these criteria, if the pointer points to a structure that contains pointers or that contains fields used to calculate addresses/pointers, then the developer is responsible for analyzing the enclave code that uses these nested pointers. The updated SDK does not address these cases, with one exception: an enclave that uses the sealing library in the SDK may take a pointer to a sealed blob as input. Code in the sealing library interprets the blob as a structure with an offset field in it. The updated SDK inserts mitigations related to the address calculated from this offset.

Real world, general cases that require developer analysis include variable-length enclave inputs with headers that include a payload length field or similar field or structure. In such cases the overall size of the input is typically provided to the enclave and enclave code checks that the nested length fields do not cause overflow — but these checks can be mis-predicted leading to speculative execution of code that is not supposed to run.

4.1.1 Bounds Check Bypass (CVE-2017-5753) Example

```
// EDL
public uint32_t enclave_function([in, size = alloc_size]tlv_t*
varlen_input, uint32_t alloc_size);
```

```
typedef struct {
     unsigned type;
     unsigned length;
     void* payload;
} tlv_t;
```

Figure 1. Enclave Function with Variable-Length Input

With the EDL code above, the SDK code will make sure that alloc_size bytes are outside the enclave and the code in the updated SDK will guard against dangerous side channels in the process. However, in the code that the developer writes, there will presumably be processing that depends on the length field of the varlen_input input. This field will not have been checked by the SDK code at all. Before the presumed length field-dependent processing in the Intel® SGX developer's code, there will presumably be a check to make sure that length is less than alloc_size. The results of this check can be mis-predicted. As a result, developers should insert an LFENCE instruction if the developer determines that the "valid length" path has a dangerous side channel when speculatively executed with an invalid length value. Of course, if performance requirements allow, the developer can simply insert an LFENCE without exhaustive analysis of the valid length path. See below.

```
//
// make sure payload is outside enclave
//
If (varlen_input.length > alloc_size) {
     // error code
     …
}
Else {
     _mm_lfence();
     //
     // valid length path
     //
     …
}
```

Figure 2. LFENCE for Enclave Variable-Length Input

_mm_lfence is the name of the LFENCE intrinsic used by some compilers. Other compilers use __builtin_ia32_lfence.

4.1.2 user_check EDL Attribute

The user_check EDL attribute instructs the SDK code to not check the associated pointer input; if used, the SDK code essentially treats the input the same way it treats an integer. As a result if user_check is used, the developer is responsible for analyzing the enclave code that uses the pointer. For user_check inputs and corresponding buffers that are supposed to be in memory outside the enclave, the following pattern can be used in cases where there appear to be dangerous side channels in the branch taken path or in cases where the performance impact of the LFENCE is considered acceptable.

```
// EDL
public uint32_t enclave_function([user_check]const uint8_t*
user_check_input, uint32_t user_check_size);
```

```
uint32_t enclave_function(const uint8_t* user_check_input,
uint32_t user_check_size)
{
     …
     //
     // make sure input buffer is outside enclave
     //
     int SGXAPI sgx_is_outside_enclave(const void *addr, size_t
     size);

     if (!sgx_is_outside_enclave(user_check_input,
     user_check_size)) {
          // error code
          …
     }
     else {
          _mm_lfence();
          …
     }
     …
}
```

Figure 3. Example of Enclave user_check Input with LFENCE

The code in Figure 3 uses sgx_is_outside_enclave, a function available in enclaves built with the SDK. As its name implies, the function checks whether the buffer specified by the pointer and size inputs is entirely outside the enclave.

If a user_check input corresponds to a structure that's supposed to be inside the enclave, an attacker can change the pointer value such that it points to the wrong memory inside the enclave. A simple check analogous to the one above will not be sufficient to mitigate this case. The enclave developer must use some other means to qualify the pointer.

4.1.3 Table/Array Indexing

For another example, consider the following victim function.

```
void victim_function(size_t x) {
     if (x < array1_size) {
          temp &= array2[array1[x] * 512];
     }
}
```

Figure 4. Dangerous Array Indexing Side Channel

If victim_function were a trusted enclave function specified in an EDL file, the SDK code could not help avoid the dangerous side channel because it would not know that the input was going to be used as an index. The developer of the enclave with a function similar to this is responsible for changing the code to mitigate the issue along the following lines:

```
void victim_function_fixed(size_t x) {
     if (x < array1_size) {
          _mm_lfence();
          temp &= array2[array1[x] * 512];
     }
}
```

Figure 5. Mitigate Dangerous Array Indexing Side Channel with LFENCE

4.1.4 sizefunc

The sizefunc EDL attribute allows a developer to specify a function that can determine the size of an input from the contents of the input itself. The function runs inside the enclave, but because it determines the size of the input, the input must be accessed before the check that makes sure the input is entirely outside the enclave. This process is susceptible to side channels. Therefore, the sizefunc attribute is being removed from the EDL immediately. In its place, developers should use the size attribute and their enclave code should always make sure that it won't go past the end of the input *before* it processes the input (see example below) or that it's not going past the end of the input *as* it processes the input.

```
size_t tcalc_size(const tlv_t* varlen_input, size_t maxlen)
{
      size_t len = sizeof(tlv_t);
      if (len > maxlen)
      {
            len = 0;
      }
      else
      {
            _mm_lfence();  //fence after check (CVE-2017-5753)
            len = varlen_input->length;
            if (len > maxlen)
            {
                  len = 0;
            }
      }
      _mm_lfence();  //fence after check (CVE-2017-5753)
      return len;
}


void ecall_no_sizefunc(tlv_t* varlen_input, size_t len)
{
       //
       // make sure code won't go past input
       // before processing
       //
      if (tcalc_size(varlen_input, len) == 0)
      {
            //record the error
            return;
      }
      //tcalc_size performs fence

      //process varlen_input
      return;
}
```

Figure 6. Another Variable-Length Input Example

4.1.5 Bounds Check Bypass Store (CVE-2018-3693) Example

The code in Figure 8 has a BCBS vulnerability. In this example, a speculative assignment to `data[user_index]`, when the bound check condition of `user_index < 8` mispredicts, allows the attacker to speculatively overwrite any variables, temporary values, or function pointers that will be called by the processor with `user_key`, which is under the attacker's control. The attacker can also speculatively modify return addresses on the stack to make the processor speculatively execute disclosure code present in the system.

```
// EDL
public int victim_function_11(int user_index, unsigned long user_key);
```

Figure 7. EDL for BCB.1 Example

```
int victim_function_11(int user_index, unsigned long user_key) {
      unsigned long data[8];
      if (user_index < 8) {
            data[user_index] = user_key;
      }
      else {
            return -1;
      }
      sort_table(data);
      return 0;
}
```

Figure 8. BCBS Vulnerability

The code in Figure 9 mitigates the vulnerability by adding an LFENCE instruction before the assignment to `data[user_index]`.

```
int victim_function_11_fixed(int user_index, unsigned long user_key) {
      unsigned long data[8];
      if (user_index < 8) {
            _mm_lfence();
            data[user_index] = user_key;
      }
      else {
            return -1;
      }
      sort_table(data);
      return 0;
}
```

Figure 9. BCBS Vulnerability Mitigated

5 Compiler Support

Microsoft* released a blog [MSFTBLOG] on January 15, 2018 describing a new compiler switch, /Qspectre, intended to help mitigate Bounds Check Bypass vulnerabilities.

Intel® Compiler versions 18.0.3 and later support compiler switches to automatically add mitigation for some Bounds Check Bypass vulnerabilities. See [INTELCOMPILERSUPPORT] for more information.

Intel® SGX developers can build their enclaves with compilers that support mitigation for Bounds Check Bypass vulnerabilities and enable the switch.

6 References

Label	Item/Link	Comment
[ANALYSIS]	Intel® Analysis of Speculative Execution Side Channels	Overview of the multiple variants along with related Intel® security features.
[CVE20175753]	CVE-2017-5753	CVE Disclosure for Bounds Check Bypass exploit
[CVE20183693]	CVE-2018-3693	CVE Disclosure for Bounds Check Bypass Store vulnerability
[GPZ]	GPZ Blog	GPZ Blog on side channel issues
[MSFTBLOG]	Spectre mitigations in MSVC	Microsoft Visual C++ has introduced compile switch that will add LFENCE instructions where it discovers Bounds Check Bypass patterns
[INTELCOMPILER SUPPORT]	Using Intel® Compilers to Mitigate Speculative Execution Side-Channel Issues	Intel® Compiler version 18.0.3 and later introduced compiling switches that add LFENCE instructions where it discovers Bounds Check Bypass patterns
[SECADV]	Speculative Execution and Indirect Branch Prediction Side Channel Analysis Method	Speculative Execution Security Advisory
[SECADVOSS2]	Speculative Execution Branch Prediction Side Channel and Branch Prediction Analysis Method	Speculative Execution Security Advisory
[SECADVOSS3]	Speculative Execution Data Cache and Indirect Branch Prediction Method Side Channel Analysis	Speculative Execution Security Advisory
[SGXDEVGUIDE]	Intel® SGX Developer Guide	Developer Guide issued with SGX SDK
[ANALYZE]	Analyze Potential Bounds Check Bypass Vulnerabilities	Describes how to analyze and mitigate common instances of bounds check bypass and bounds check bypass store.