



What to Do When It is Already Too Late ?

Crashdumps for Embedded Systems

Christoph Sterz
christoph.sterz@kdab.com

Embrace the Fail

Content

- The nature of a crash and where to intercept
- Working with core dumps
- Signal handlers and watchdogs
- My serving suggestion:
 - Yocto, and...
 - Google Breakpad, and...
 - Sentry
- On collecting crashdumps in production

Scope of *Crashes* in this Talk

- Crashes mostly in C/C++
- On Embedded Linux
- Crashes induced from the inside and from the outside
- No Kernelpanics, etc.
- SW-Devs' Assumption #1: Hardware just works

Crashes in Development And Production

- **Dev Environment on Embedded Devices**

- All Syms / (Remote) GDBs
- Fullsize dumps
- EvalBoards
- Small Testing Surface

- **In Production**

- Slim Images
- Slim Dumps / Reduced Bandwidth
- (Often more limited) production hardware, **no storage**
- Large Testing Surface

Crashes in Development And Production

• Dev Environment on Embedded Devices

- All Syms / (Remote) GDBs
- Fullsize dumps
- EvalBoards
- Small Testing Surface

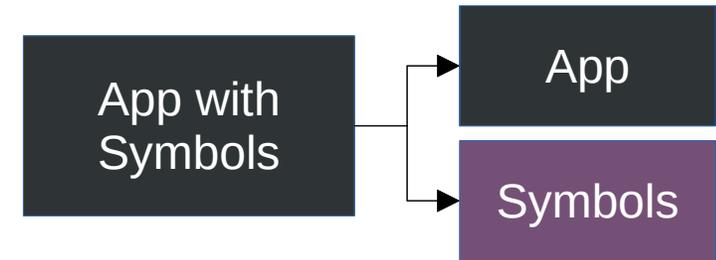
Boils down to
storage <vs.> no storage

• In Production

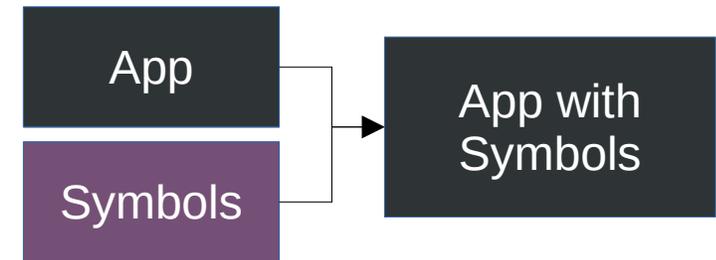
- Images
- Dumps / Reduced Bandwidth
- (Often more limited) production hardware, **no storage**
- Large Testing Surface

Crashdumps and Symbols

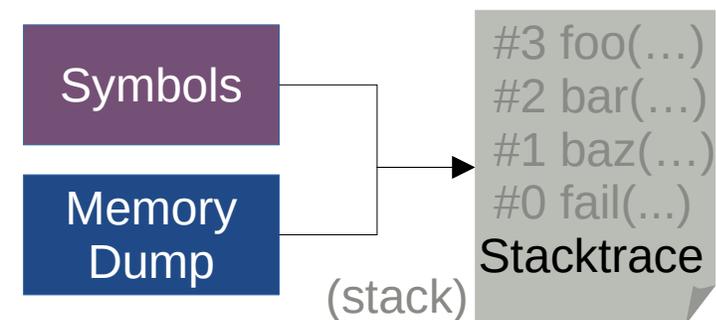
- Symbols are needed:
 - To make addresses readable for humans
 - To reconstruct the contents of the Stack
 - To infer Line Numbers
- You will get symbols with -g
- Symbols are *independent* of optimization (-g, -O2)
- Symbols are huge



stripping

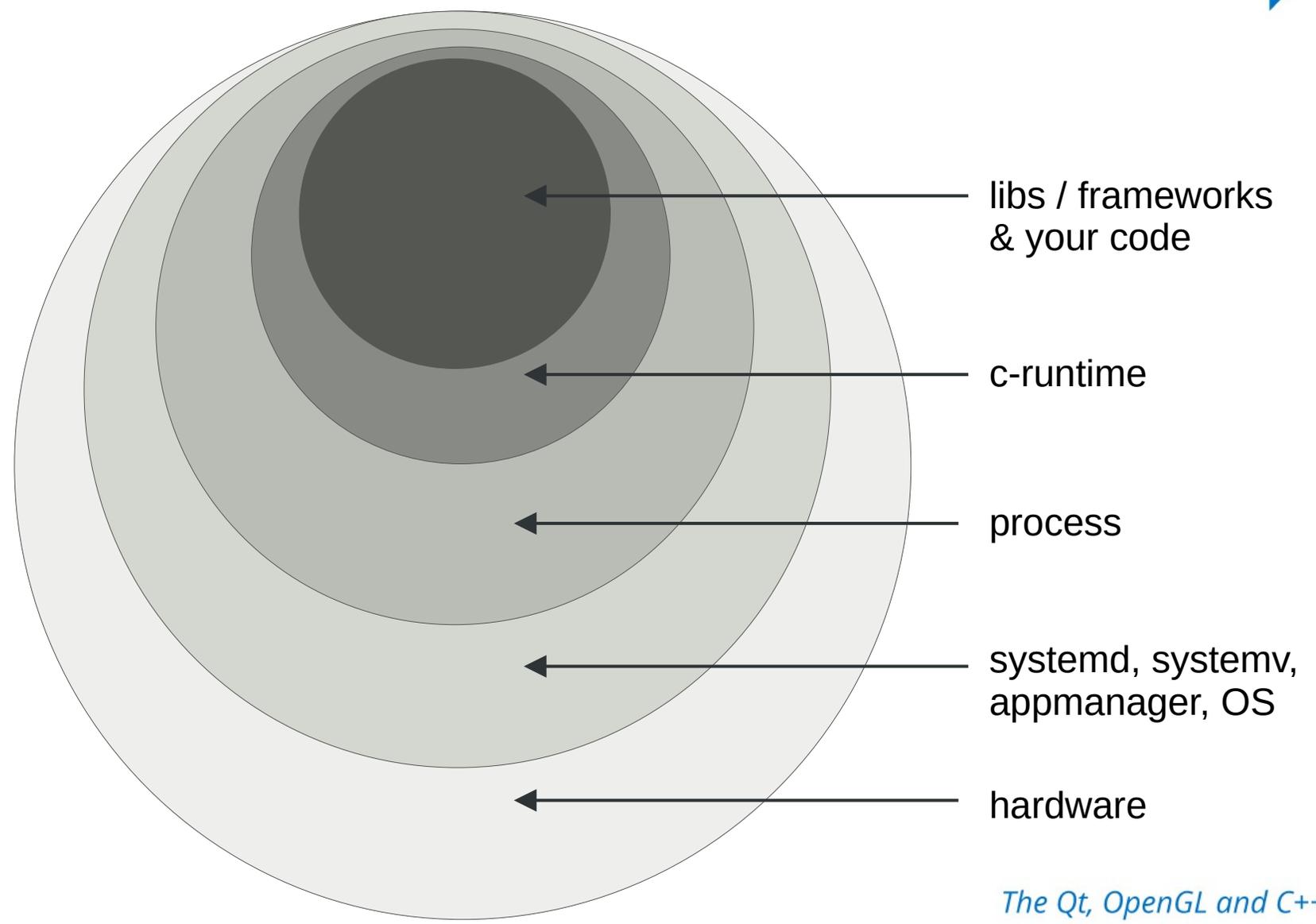


unstripping



(stack)
unwinding

Code is embedded in many execution contexts.



4 Bytes of Core Memory : Arduino Module

Coredumps

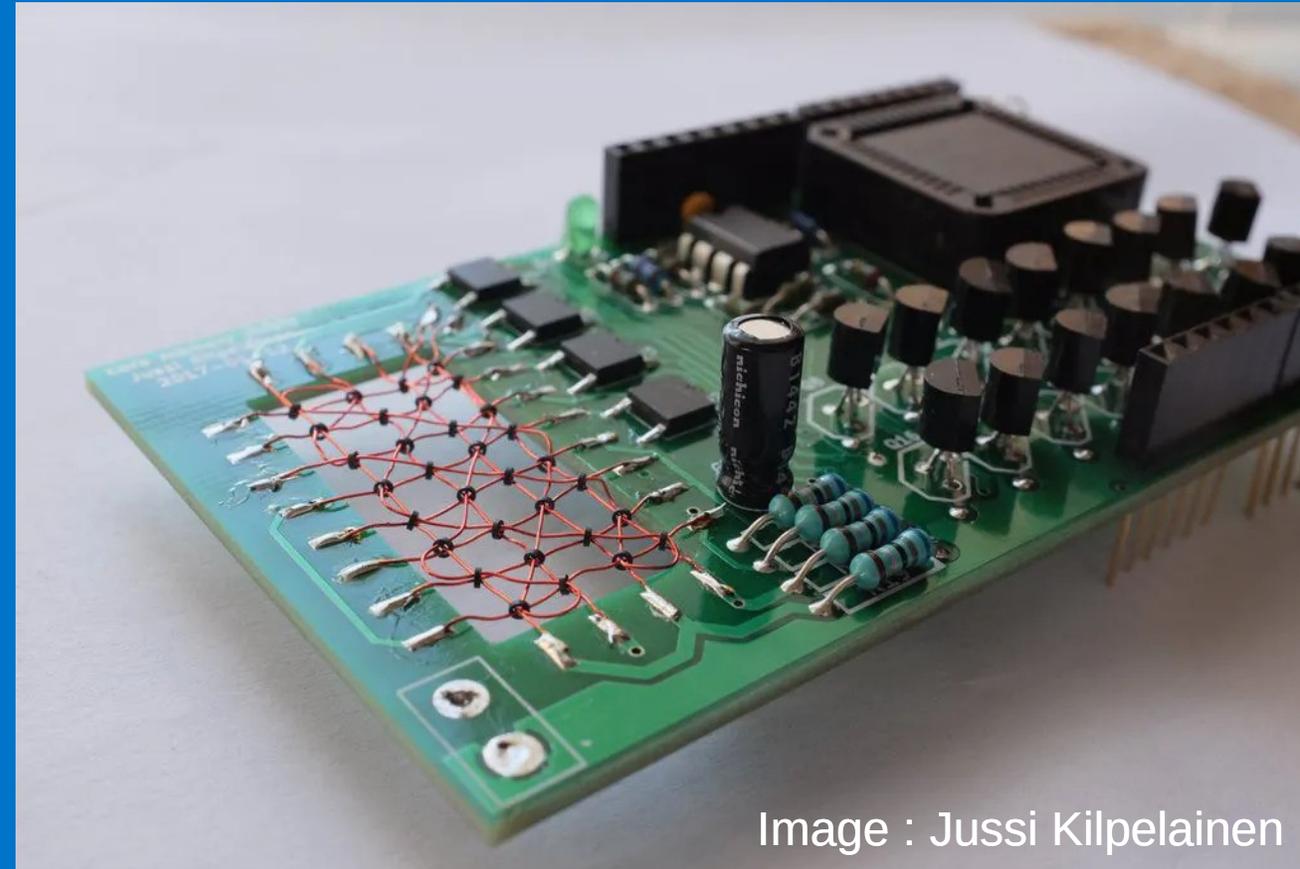


Image : Jussi Kilpelainen

What do Core dumps Look Like?

- TODO

Prerequisites

- CONFIG_COREDUMP enabled in the Kernel
- Executable must be readable (guess why?)
- Process must have permissions to write the core

- You need enough space to store it
- You need enough Bandwidth to transfer it

Enable by setting limits

```
root@imx6ul-var-dart:~# ulimit -a
core file size          (blocks, -c) 0
data seg size          (kbytes, -d) unlimited
scheduling priority    (-e) 0
```



```
root@imx6ul-var-dart:~# ulimit -c unlimited
root@imx6ul-var-dart:~# ulimit -a
core file size          (blocks, -c) unlimited
data seg size          (kbytes, -d) unlimited
scheduling priority    (-e) 0
file size              (blocks, -f) unlimited
pending signals        (-i) 3938
```

/proc/sys/kernel/core_pattern

Path containing %P %u %g %s %t %c %h

PID

uid

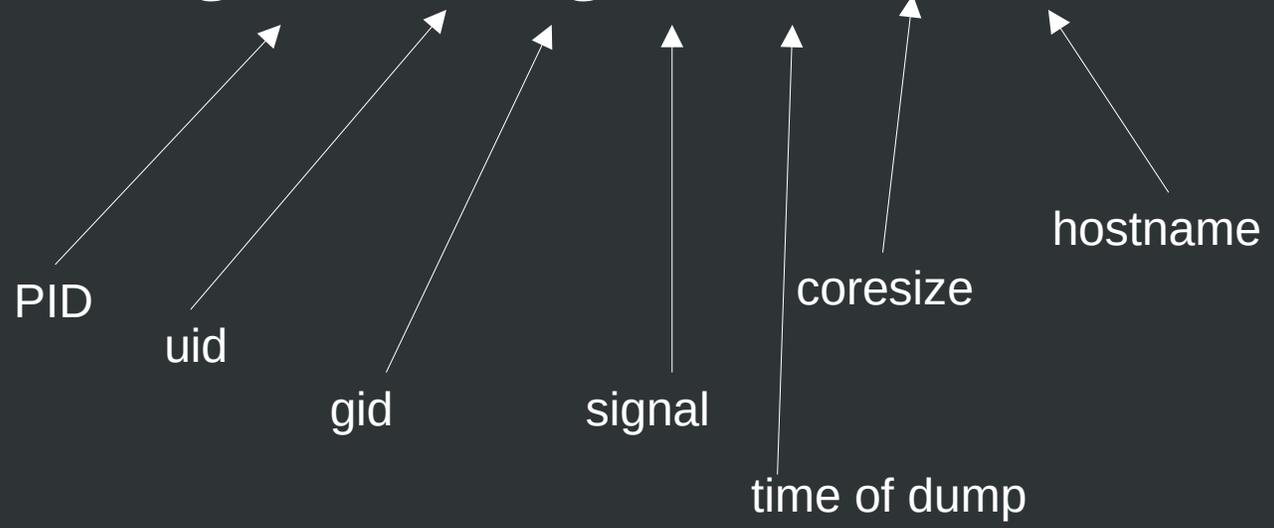
gid

signal

time of dump

coresize

hostname



CoreDumps: Did you know?

- You can **advise** memory pages to be excluded from Coredump
 - Use **advise** with **MADV_DONTDUMP** flag
- You can pipe coredumps to stdin of another process
 - Make your corePattern start with a | character, followed by the receiving process
 - Systemd coredumpctl does it |**/usr/lib/systemd/systemd-coredump**
- GDBs **gcore** can create a core of a running process

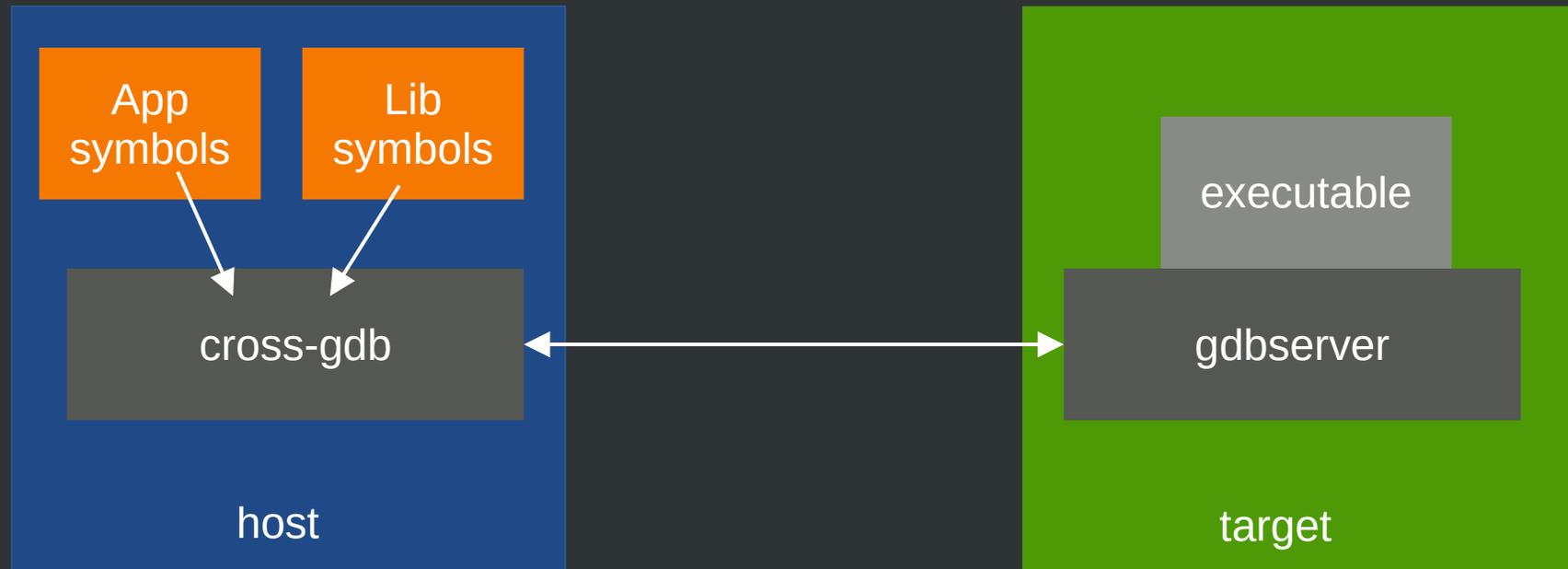
Development: have GDB on your target!

- At the Development Stage, just have a gdb on the target
- Find a way to store the coredump
- If you get a Crash producing a coredump, rejoin symbols:
 - Use the elfutils bin **eu-unstrip** **<executable>** **<symbols>**
 - Repeat for all relevant libraries you need for heap / stack
- Its a bit tedious, its worth it, if you need heap information
- If no heap is needed, there are better ways

You need it all for Cross-Platform CoreDumps

- A cross-toolchain GDB on your Desktop
- The exact same executable that crashed (/w symbols)
- Symbols for all relevant libs when it crashed
- The core file
- Optionally /proc/kallsyms from the target
- Carefully feed SDK-Paths and Libs and see if you get a stack

GDB-Server: A Hybrid



Poor-Man's SlimDump: Backward-Cpp

By François-Xavier Bourlet, @bombela

- The Pitch

```
christoph@mareike /tmp/backward-cpp/build $ ./test_suicide  
Segmentation fault (core dumped)
```

Tired of seeing this ?

```
230:         } else {
#3 Source "/tmp/backward-cpp/test/_test_main.cpp", line 140, in run_test [0x55e66a01cd0c]
    138:     pid_t child_pid = fork();
    139:     if (child_pid == 0) {
> 140:         exit(static_cast<int>(test.run()));
    141:     }
    142:     if (child_pid == -1) {
    143:         error(EXIT_FAILURE, 0, "unable to fork");
#2 Source "/tmp/backward-cpp/test/test.hpp", line 92, in run [0x55e66a01d143]
    90:     TestStatus run() {
    91:         try {
> 92:             do_test();
    93:             return SUCCESS;
    94:         } catch (const AssertFailedError &e) {
    95:             printf("!! %s\n", e.what());
#1 Source "/tmp/backward-cpp/test/suicide.cpp", line 40, in do_test [0x55e66a00e940]
    37:     *ptr = 42;
    38: }
    39:
> 40: TEST_SEGFAULT(invalid_write) { badass_function(); }
    41:
    42: int you_shall_not_pass() {
    43:     char *ptr = (char *)42;
#0 Source "/tmp/backward-cpp/test/suicide.cpp", line 37, in badass_function [0x55e66a00e92a]
    35: void badass_function() {
    36:     char *ptr = (char *)42;
> 37:     *ptr = 42;
    38: }
    39:
    40: TEST_SEGFAULT(invalid_write) { badass_function(); }
```

Then Try this :)

Segmentation fault (Address not mapped to object [0x2a])

!! signal (11) Segmentation fault

christoph@mareike /tmp/backward-cpp/build \$ █

BackwardCpp

- This is in the Development-Situation category
 - Include a header + 1 Line of initialization
 - Symbols are necessary in build (-g), fat binaries
 - Does stack unwinding in the signal handlers
 - Requires the source code to print it
 - Can be heavily customized
- This is great for Development!

The Sanitizers can help you as well.

Crash output of an executable, instrumented with the gcc/clang address sanitizer

```
AddressSanitizer:DEADLYSIGNAL
=====
==46184==ERROR: AddressSanitizer: SEGV on unknown address 0x00000000002a (pc 0x5
==46184==The signal is caused by a WRITE memory access.
==46184==Hint: address points to the zero page.
#0 0x555c1c0f1fd4 in ManualBrewing::setPump
#1 0x555c1c4d7cf5 in ManualBrewing::qt_metacall
#2 0x7f9e58707d5f in QQmlPropertyPrivate::write

#3 0x7f9e58633078 in QV4::QObjectWrapper::setProperty
#4 0x7f9e58633aa8 in QV4::QObjectWrapper::setQmlProperty

#5 0x7f9e58633c46 in QV4::QObjectWrapper::virtualPut
#6 0x7f9e585fe52a in QV4::Object::virtualResolveLookupSetter
#7 0x7f9e5864c808 (/usr/lib/libQt5Qml.so.5+0x1b0808)
#8 0x7f9e5865068e (/usr/lib/libQt5Qml.so.5+0x1b468e)
#9 0x7f9e585ead2d in QV4::Function::call
#10 0x7f9e58766915 in QQmlJavaScriptExpression::evaluate
#11 0x7f9e5871962c in QQmlBoundSignalExpression::evaluate
#12 0x7f9e58719b10 (/usr/lib/libQt5Qml.so.5+0x27db10)
#13 0x7f9e5874a00c in QQmlNotifier::emitNotify
#14 0x7f9e57fb5904 (/usr/lib/libQt5Core.so.5+0x2ec904)
#15 0x7f9e586f76ea in QQmlVMEMetaObject::metaCall
#16 0x7f9e5874a56d (/usr/lib/libQt5Qml.so.5+0x2ae56d)
#17 0x7f9e5862f946 (/usr/lib/libQt5Qml.so.5+0x193946)
#18 0x7f9e58631f39 in QV4::QObjectMethod::callInternal
#19 0x7f9e5865f2f9 in QV4::Runtime::CallPropertyLookup::call
#20 0x7f9e399d9af1 (/memfd:JITCode:QtQml (deleted)+0xaf1)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV
==46184==ABORTING
```

No Symbols?, Unwinding Fails?
You can still resort to:

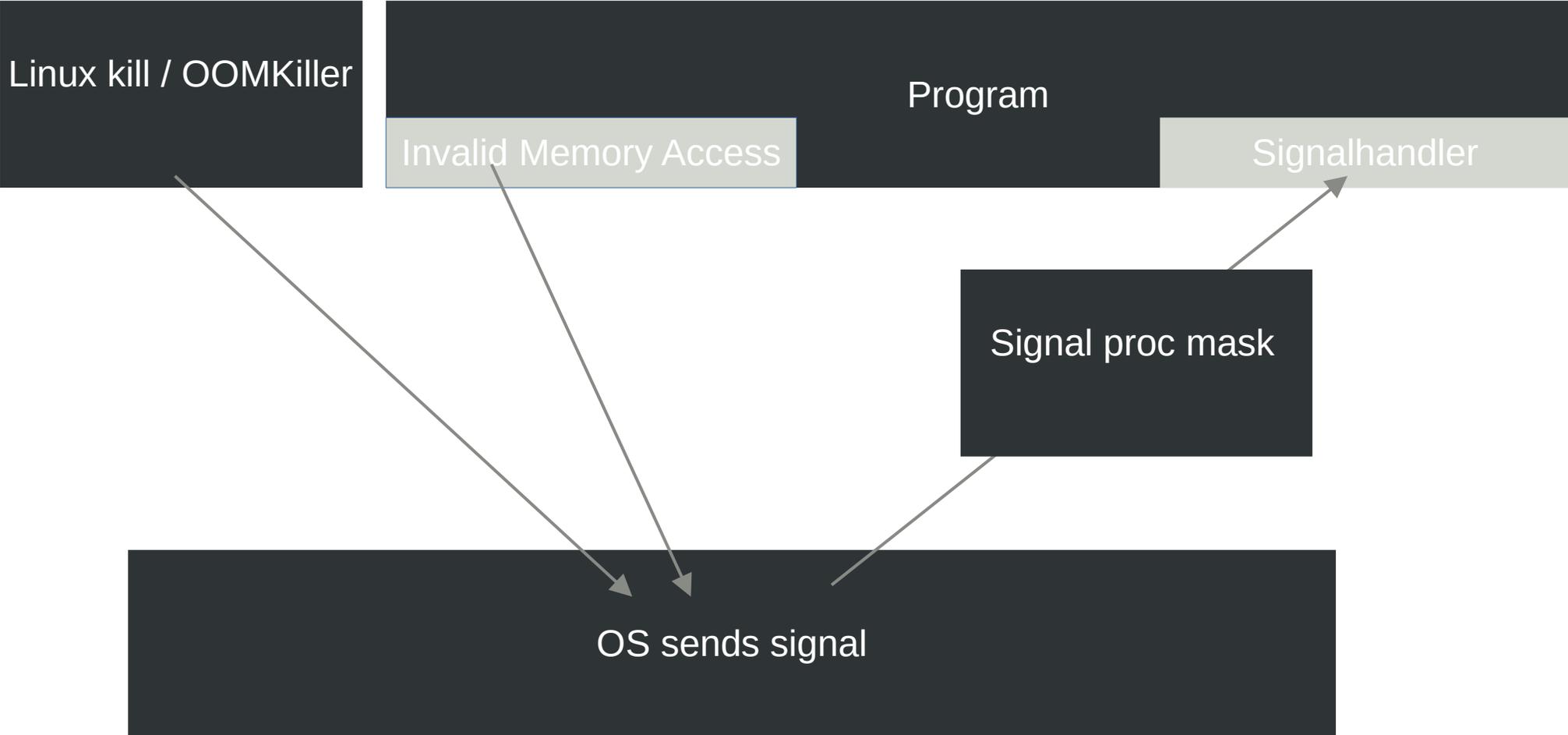
»*Desperate-Stack-Reading*«

Printing raw stack memory, garnished with symbols take everything with teaspoons of salt

```
(gdb) set print asm-demangle on
(gdb) x/300a $sp
0x7fffffff8db0: 0x3000000009      0x7fffffff8dc0
0x7fffffff8dc0: 0x0              0x0
0x7fffffff8dd0: 0x7fffffff8e70  0x7ffff694bd60 <QqmlPropertyPrivate::write(QObject*, QqmlPropertyData const&, QVariant const&, QqmlContextData*, QFlags<QqmlPropertyData::WriteFlag>)+448>
0x7fffffff8de0: 0x555555628a301  0x1
0x7fffffff8df0: 0x2              0x466a05fb69427e00
0x7fffffff8e00: 0x3              0x5555556272508
0x7fffffff8e10: 0x2              0x7fffffff8f10
0x7fffffff8e20: 0x555555f208e0  0x5555555e38db <ManualBrewing::qt_metacall(QMetaObject::Call, int, void**)+139>
0x7fffffff8e30: 0x2              0x200000010
0x7fffffff8e40: 0x7fffffff8ff0  0x1
0x7fffffff8e50: 0x555555f208e0  0x7ffff694bd60 <QqmlPropertyPrivate::write(QObject*, QqmlPropertyData const&, QVariant const&, QqmlContextData*, QFlags<QqmlPropertyData::WriteFlag>)+448>
0x7fffffff8e60: 0x55555562d6a01  0x466a05fb69420001
0x7fffffff8e70: 0x0              0x1
0x7fffffff8e80: 0x7ffff01316b8  0x7ffff0131700
0x7fffffff8e90: 0x7ffff0131708  0x7ffff01316e8
0x7fffffff8ea0: 0x7ffff01316b8  0xfe
0x7fffffff8eb0: 0x7ffff01316b8  0x7ffff6842ec6 <QV4::Object::insertMember(QV4::StringOrSymbol*, QV4::Property const*, QV4::PropertyAttributes)+70>
0x7fffffff8ec0: 0x0              0xf68427c8
0x7fffffff8ed0: 0x7ffeffffffff  0x466a05fb69427e00
0x7fffffff8ee0: 0x7f00ffffff    0x555555d7d000
0x7fffffff8ef0: 0x7ffff0131708  0x555555d7d000
0x7fffffff8f00: 0x7ffff0131700  0x466a05fb69427e00
0x7fffffff8f10: 0x7fffffff8ff0  0x0
0x7fffffff8f20: 0x7fffffff8ed0  0x7fffffff8ec0
0x7fffffff8f30: 0x555555d7d000  0x466a05fb69427e00
0x7fffffff8f40: 0x555555f208e0  0x7ffff01316b8
0x7fffffff8f50: 0x555555d7d000  0x7ffff0131610
0x7fffffff8f60: 0x555555f208e0  0x7fffffff8ff0
0x7fffffff8f70: 0x5555556272508 0x7ffff6877079 <QV4::QObjectWrapper::setProperty(QV4::ExecutionEngine*, QObject*, QqmlPropertyData*, QV4::Value const&)+2601>
0x7fffffff8f80: 0x7ffff01316b8  0x7ffff01316c8
0x7fffffff8f90: 0x7ffff01316d0  0x7fffffff9128
0x7fffffff8fa0: 0x1              0x555555ee4dc0
```

Signal Handlers can act
when its already too late

But they are limited
– use them with care!



Things **not allowed** in the SignalHandler

- Heap allocations are forbidden from this point on
- One is only permitted to execute “safe” operations
 - That is basically everything that does not allocate
 - Check **man signal-safety** for it
 - Code looks much like pure C-Code then
- Be hyper-careful of Crashes in Crash Handlers.
You have been warned :)

Things **allowed** in the SignalHandler

- Start new processes (wow!)
- Obviously reading heap memory
- Again, everything that man signal-safety allows you to do
- Send signal to self

Signalhandlers / Crashhandlers look much like plain C Code

```
3
4 static bool dumpCallback(const google_breakpad::MinidumpDescriptor& descriptor,
5                          void* context, bool succeeded) {
6
7     // start new process to turn of pump, heating, etc
8     // fork returns 0 for the child
9     if (fork()) {
10        printf("App Crashed. Dump can be found at: %s\n", descriptor.path());
11        const auto& stack = static_cast<ScreenManager*>(context)->getStack();
12        char* filename = strcat(const_cast<char*>(descriptor.path()), ".additional")
13        int screenStackTrace = open(filename, O_CREAT | O_WRONLY, 0644);
14        char buf[255];
15        const char* start = "{\\"Screenstack\\":\\"";
16        write(screenStackTrace, start, strlen(start));
17        for (const auto& entry : stack) {
18            snprintf(buf, sizeof (buf), "%s ", entry.toStdString().c_str());
19            write(screenStackTrace, buf, strlen(buf));
20        }
21        const char* end = "\\}";
22        write(screenStackTrace, end, strlen(end));
23        close(screenStackTrace);
24    } else {
25        char* const argv[] = {(char*)"stop.sh", NULL};
26        execve("/opt/crash/SystemCrashHandler.sh", argv, NULL);
27    }
28    return succeeded;
29 }
```

Cool Things possible in the Signal Handler

- Check out the KDABs *QML stack trace dumper*[1].
 - Actually unsafe, because allocates
 - but worth the gamble, its too late anyways, right?
- TODO More

[1]<https://github.com/KDAB/KDToolBox/tree/master/qt/qml/QmlStackTraceHelper>

Watchdog-like processes can assist

It makes sense to have Watchdogs out of the main execution Context

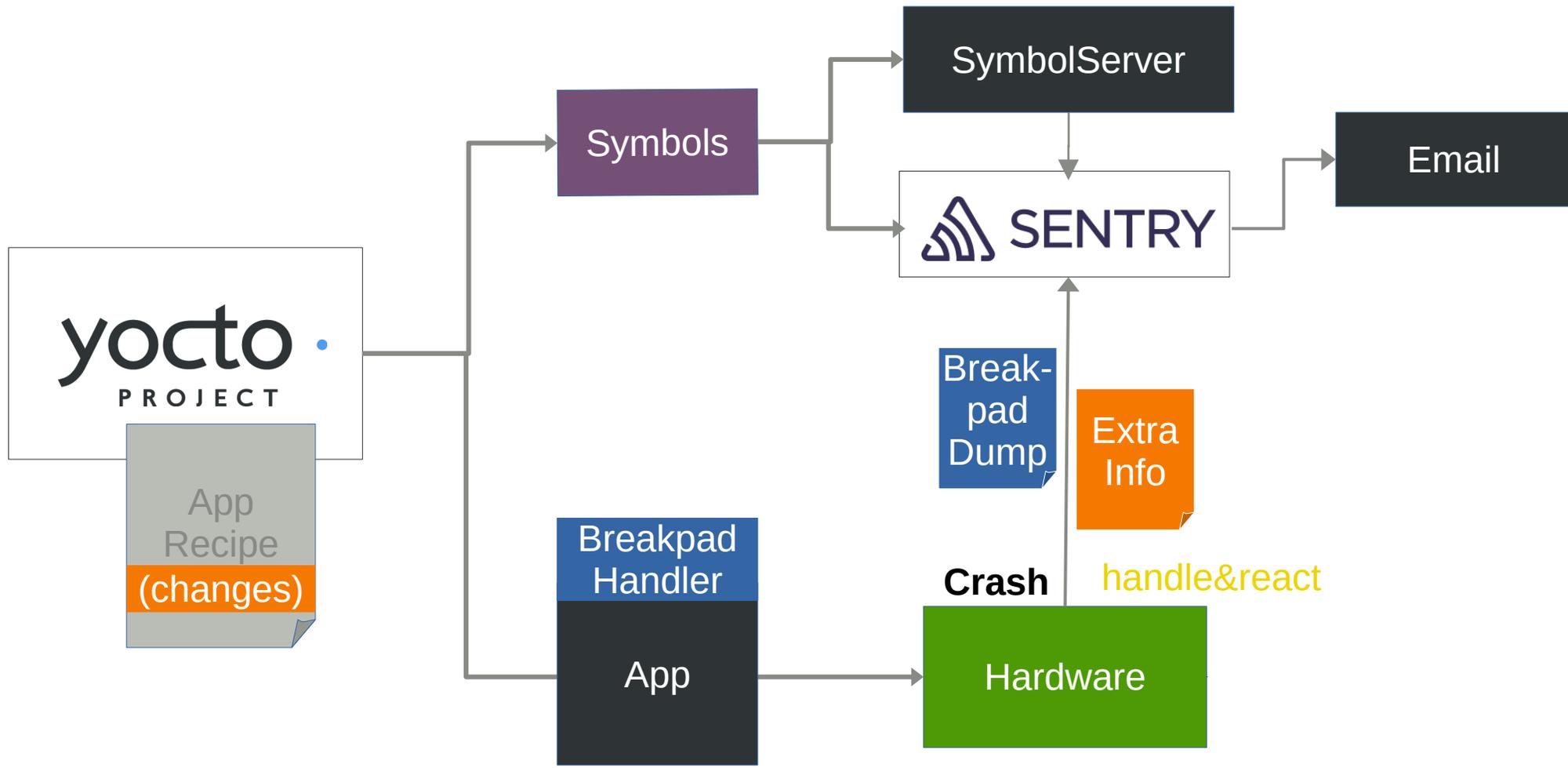
- There is not only Crashes, but also infinite loops
 - Idea: Reset the watchdog periodically, such that infinite loops are detected
- It can make a lot of sense to inject SIGSEGV & friends from the outside
- A stacktrace will still be produced and Analysis is possible

The OOM killer

- Most famous external source of an unwanted termination
- Stackdumps are of limited use in Analysis here
- Sends SIGKILL in rare cases also SIGTERM
 - Here, app dumps are a bit less interesting
 - Collect specifically the system-wide memory situation in this case

In Practice: Google Breakpad + Sentry + Yocto

General Architecture



Integrate Google Breakpad into Yocto

- Breakpad Recipes included in meta-oe/recipes-devtools
 - Creates all cross-tools needed
 - Creates the header-only library needed for the custom Signal Handler
 - Provides a yocto bbclass to be added to your app recipe
 - This then splits out symbols before app-binary is stripped by yocto
- Extras can be added in your individual app recipe



For the **other libs**, I use the yocto-built SDK, it contains split debug symbols in .debug folders

```
christoph@mareike /tmp $ ls $SDKTARGETSYSROOT/usr/lib/.debug/
e2initrd_helper          libgstinsertbin-1.0.so.0.1404.0  libQt53DInput.so.5.12.2
libarchive.so.13.3.3     libgstisoff-1.0.so.0.1404.0     libQt53DLogic.so.5.12.2
libasm-0.175.so         libgstmpegts-1.0.so.0.1404.0    libQt53DQuickAnimation.so.5.12.2
libasound.so.2.0.0      libgstnet-1.0.so.0.1404.0       libQt53DQuickExtras.so.5.12.2
libatomic.so.1.2.0      libgstpbutils-1.0.so.0.1404.0   libQt53DQuickInput.so.5.12.2
libbluetooth.so.3.18.16 libgstphotography-1.0.so.0.1404.0 libQt53DQuickRender.so.5.12.2
libbtrfs.so.0.1         libgstplayer-1.0.so.0.1404.0    libQt53DQuickScene2D.so.5.12.2
libbtrfsutil.so.1.0.0   libgststreamer-1.0.so.0.1404.0  libQt53DQuick.so.5.12.2
libbz2.so.1.0.6         libgststriff-1.0.so.0.1404.0    libQt53DRender.so.5.12.2
libcairo-gobject.so.2.11400.12 libgststrtp-1.0.so.0.1404.0     libQt5Bluetooth.so.5.12.2
libcairo-script-interpreter.so.2.11400.12 libgstrtsp-1.0.so.0.1404.0     libQt5Charts.so.5.12.2
```

Example for file **libQt5Core.so** : It is important, that debug info is present

```
christoph@mareike /tmp $ file $SDKTARGETSYSROOT/usr/lib/.debug/libQt5Core.so.5.12.2
/home/christoph/KDAB/Braumeister/sdk/sysroots/cortexa7t2hf-neon-fslc-linux-gnueabi/usr/lib/.debug/libQt5Core.so.5.12.2: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (GNU/Linux), dynamically linked, BuildID[sha1]=ea22fbb2d6efcba010ba2cf02739cbe31cff7c7a, for GNU/Linux 4.11.0, with debug_info, not stripped
```

... from there it is uploaded like all syms with **sentry-cli**

```
christoph@mareike /tmp $ sentry-cli --url https://sentry10.kdab.com/ --auth-token '4617bd' upload-dif -o kdab -p Braumeister $SDKTARGETSYSROOT/usr/lib/.debug/libQt5Qml.so.5.12.2
> Found 1 debug information file
> Prepared debug information file for upload
> Uploaded 1 missing debug information file
> File upload complete:

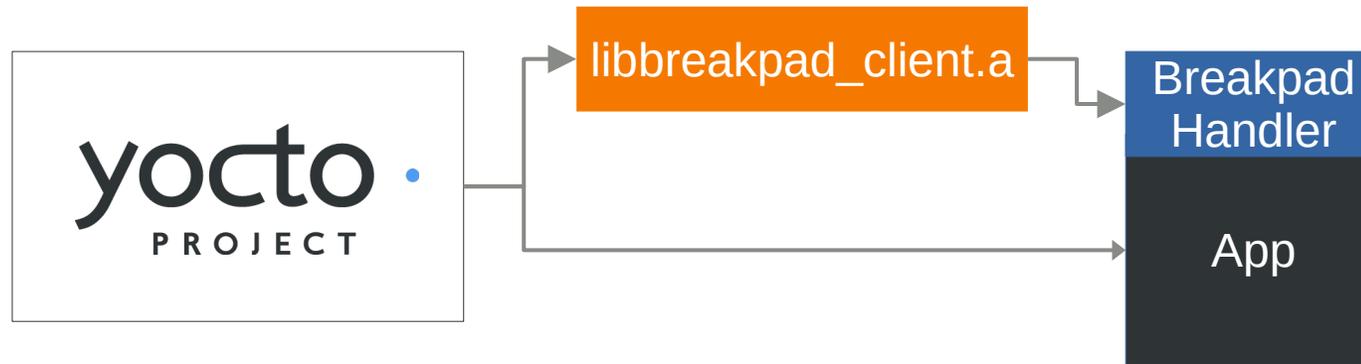
  PENDING 286179fe-faec-82c0-7af9-97c1d4ad120d (libQt5Qml.so.5.12.2; arm debug companion)
christoph@mareike /tmp $
```

More Infos on Googles Breakpad

- Uses Minidumps
 - Originally envisioned by Microsoft
 - Similar to slim cores, but way smaller (around 20KiB)
 - Cross-platform (unix cores don't work on Windows, settled on minidump)
 - Splitting command: **dump_syms executable > /path/to/destination.syms**
- Minidump comes with some useful tools
 - **minidump_stackwalk**: Re-Combine Minidump+App+Symbols → get a stack
 - **minidump-2-core**: Converts dump to gdb-readable format
 - and more...

Integrate Breakpad in your Code

- Breakpads library and headers are included in the new SDK when using it in any of your recipes
- Only 2 extra lines in main() are necessary to register
- Of course you can do more in your custom Handler



Register the handler in your main(), pass any variables to be used

```
33
34 int main(int argc, char *argv[])
35 {
36 //...
37
38 #ifndef TARGET
39     google_breakpad::MinidumpDescriptor descriptor("/home/root/crashreports/");
40     google_breakpad::ExceptionHandler eh(descriptor, NULL, dumpCallback, screenManager, true, -1);
41 #endif
42
```

Register Breakpad
Handler

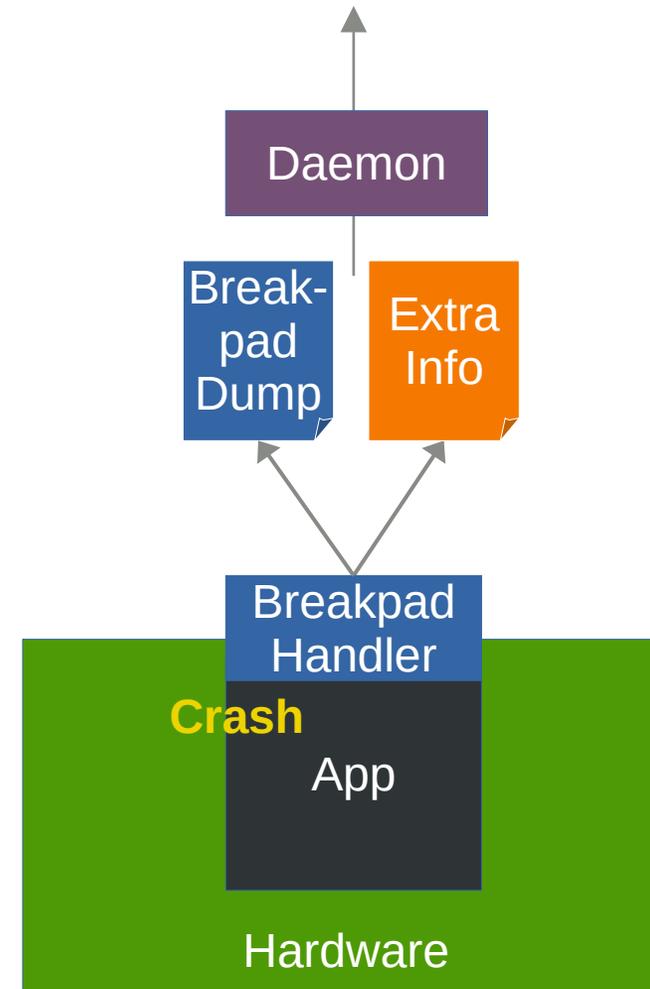
Include Breakpad Header, Handle crashes and write extra information

```
1 #ifdef Target
2 //Breakpad Crashreporter
3 #include "client/linux/handler/exception_handler.h"
4
5 static bool dumpCallback(const google_breakpad::MinidumpDescriptor& descriptor,
6                          void* context, bool succeeded) {
7
8     // start new process to turn of pump, heating, etc
9     // fork returns 0 for child
10    if (fork()) {
11        printf("App Crashed. Dump can be found at: %s\n", descriptor.path());
12        const auto& stack = static_cast<ScreenManager*>(context)->getStack();
13        char* filename = strcat(const_cast<char*>(descriptor.path()), ".additional");
14        int screenStackTrace = open(filename, O_CREAT | O_WRONLY, 0644);
15        char buf[255];
16        const char* start = "{\\"Screenstack\\":\\"";
17        write(screenStackTrace, start, strlen(start));
18        for (const auto& entry : stack) {
19            snprintf(buf, sizeof (buf), "%s ", entry.toStdString().c_str());
20            write(screenStackTrace, buf, strlen(buf));
21        }
22        const char* end = "\\"}";
23        write(screenStackTrace, end, strlen(end));
24        close(screenStackTrace);
25    } else {
```

Write Extra
Information

Sending the Information

- Let a daemon check the crash folder for crashes
 - Not known if device has connectivity
 - Daemon checks periodically if a minidump is available
 - If allowed in the User-settings, Info is uploaded to the sentry server
- For now, no logs are uploaded, maybe in the future...



Snip from the Crashdaemon,
A file watcher looks for crash data and uploads it, when possible
Extra tag info is garnished for sentry

```
9
10 QFile additional(s_crashReportPath + filename + ".additional");
11 additional.open(QIODevice::ReadOnly);
12 const QByteArray tags = additional.readAll();
13 additional.close();
14
15 QFile crashreport(s_crashReportPath + filename);
16 crashreport.open(QIODevice::ReadOnly);
17 const QByteArray report = crashreport.readAll();
18 crashreport.close();
19
20 QHttpMultiPart* multipart = new QHttpMultiPart(QHttpMultiPart::FormDataType);
21
22 QHttpPart reportPart;
23 reportPart.setHeader(QNetworkRequest::ContentTypeHeader, QVariant("application/octet-stream"));
24 reportPart.setHeader(QNetworkRequest::ContentDispositionHeader, QVariant("form-data; name=\"upload_file_minidump\"; filename=\"" + filename + "\""));
25 reportPart.setBody(report);
26
27 QHttpPart jsonPart;
28 jsonPart.setHeader(QNetworkRequest::ContentDispositionHeader, QVariant("form-data; name=\"sentry\""));
29 jsonPart.setBody("{\"tags\": " + tags + "}");
30
31 multipart->append(reportPart);
32 multipart->append(jsonPart);
33
34 const QUrl uploadUrl(QUrl("https://sentry10.kdab.com/api/3/minidump?sentry_key=13e8"));
35 QNetworkRequest request(uploadUrl)
36 //...
```

The Result

Issues (15) Sort by: Events Custom Search

| | GRAPH: | 24h 30d | EVENTS | USERS | ASSIGNEE |
|---|--------|----------------------|--------|-------|----------|
| <input type="checkbox"/> __GI___errno_location in errno-loc.c Unhandled Fatal Error: SIGSEGV /0x00000000 BRAUMEISTER-T 15 days ago – a month old | | | 4 | 0 | |
| <input type="checkbox"/> _int_realloc in malloc.c Unhandled Fatal Error: SIGSEGV /0x00000000 BRAUMEISTER-1A 17 days ago – 17 days old | | | 3 | 0 | |
| <input type="checkbox"/> BrewTimer::isEnabled in brewtimer.cpp Unhandled Fatal Error: SIGSEGV /0x00000000 BRAUMEISTER-18 25 days ago – 25 days old | | //Categorized | 3 | 0 | |
| <input type="checkbox"/> _fini Unhandled Fatal Error: SIGSEGV /0x00000000 BRAUMEISTER-Y a month ago – a month old | | | 2 | 0 | |
| <input type="checkbox"/> <unknown> Unhandled Fatal Error: SIGSEGV /0x00000000 BRAUMEISTER-5 17 days ago – 2 months old | | | 2 | 0 | |
| <input type="checkbox"/> QVector<T>::erase in qrefcount.h Unhandled Fatal Error: SIGSEGV /0x00000000 BRAUMEISTER-1G 17 days ago – 17 days old | | | 1 | 0 | |



Unknown User



Linux
Version: 4.14.78-mx6ul+gaecdf94



Unknown Device
Arch: arm



TAGS

//Extra Information

//Breakpad Information

Screenstack startmenu.qml recipes/my_recipes.qml

handled no level fatal mechanism minidump os Linux 4.14.78-mx6ul+gaecdf94 os.name Linux

Thread #346: QVector<T>::erase (most recent call first)

Full Raw

id 346 was active false errored yes

SIGSEGV /0x00000000

Fatal Error: SIGSEGV /0x00000000

mechanism minidump handled false

app +0x06521e QVector<T>::erase (qrefcount.h:68)

| | | | | | | | | |
|-----------|----|--------------------|----|--------------------|-----|--------------------|-----|--------------------|
| registers | r0 | 0x0000000000000010 | r1 | 0x000000002504b14 | r10 | 0x0000000000000010 | r12 | 0x000000000000000c |
| | r2 | 0x000000002504b18 | r3 | 0x0000000000000001 | | 0x000000002504b18 | r5 | 0x000000002504b14 |

Show More

//Symbols available

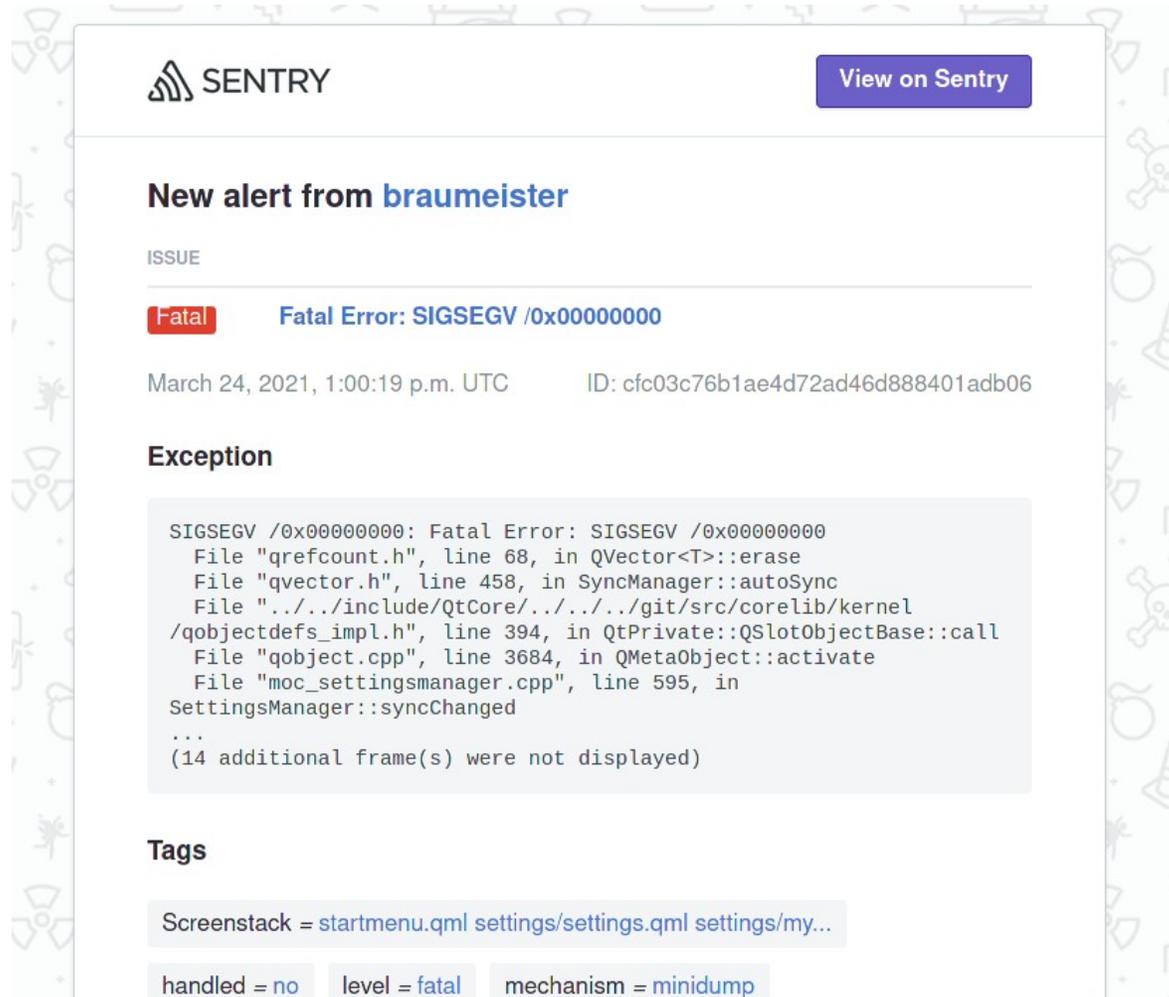
| | | |
|-------------------|-----------|---|
| app | +0x062c66 | SyncManager::autoSync (qvector.h:458) |
| libQt5Core.so.... | +0x1e0806 | QtPrivate::QSlotObjectBase::call (../../include/QtCore/../../git/src/corelib/kernel/qobjectdefs_impl.h:394) |
| libQt5Core.so.... | +0x1e0806 | QMetaObject::activate (qobject.cpp:3684) |
| app | +0x071594 | RecipeReader::addRecipe (recipereader.cpp:50) |
| app | +0x06387a | SyncManager::receiveRecipeResponse (syncmanager.cpp:610) |
| app | +0x064224 | SyncManager::receiveResponse (syncmanager.cpp:534) |
| libQt5Core.so.... | +0x1e0806 | QtPrivate::QSlotObjectBase::call (../../include/QtCore/../../git/src/corelib/kernel/qobjectdefs_impl.h:394) |
| libQt5Core.so.... | +0x1e0806 | QMetaObject::activate (qobject.cpp:3684) |

//Symbols not available

| | | |
|-------------------|-----------|-------------|
| libQt5Network.... | +0x031307 | <unknown> ⚠ |
| libQt5Network.... | +0x031e41 | <unknown> ⚠ |

+ experts

Mail



The image shows a screenshot of an email alert from Sentry. The header includes the Sentry logo and a 'View on Sentry' button. The main content is a 'New alert from braumeister' with the following details:

- ISSUE**
- Fatal** Fatal Error: SIGSEGV /0x00000000
- March 24, 2021, 1:00:19 p.m. UTC ID: cfc03c76b1ae4d72ad46d888401adb06
- Exception**

```
SIGSEGV /0x00000000: Fatal Error: SIGSEGV /0x00000000
  File "qrefcount.h", line 68, in QVector<T>::erase
  File "qvector.h", line 458, in SyncManager::autoSync
  File ".../include/QtCore/../../../../git/src/corelib/kernel
/qobjectdefs_impl.h", line 394, in QtPrivate::QSlotObjectBase::call
  File "qobject.cpp", line 3684, in QMetaObject::activate
  File "moc_settingsmanager.cpp", line 595, in
SettingsManager::syncChanged
...
(14 additional frame(s) were not displayed)
```

Tags

Screenstack = startmenu.qml settings/settings.qml settings/my...

handled = no level = fatal mechanism = minidump

More about sentry

- FOSS Dump Collector
- Sorts Crashes by Stack, configurable
- Supports many DumpFormats
- Supports external Symbol Servers

- ~30€/mo or self hosted

GDPR?
<Im not a lawyer>, but...

On uploading crash(=user)data

- We run it for development / staging / testing only
- If Production is involved, make it opt-In for users
- Practically, stack information might contain all information
- *If dumps are anonymized and users know for what purpose the data is collected, one should be fine*
... </but Im not a lawyer>

- #1 Overall there is still much to do,
when its already too late
- #2 I showed you classic ways in therory
and one way I like in practice
- #3 Invest in learning from your crashes
—it is worth it.

End Of Talk!

I will answer all questions, AMA!