KAATUMISILMOITUSTEN KERÄÄMISEN JA ANALYSOINNIN AUTOMATISOINTI

Ville Kilpeläinen

Opinnäytetyö
Huhtikuu 2018
Tieto- ja viestintätekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikka
Ohjelmistotekniikka

KILPELÄINEN VILLE:
Kaatumisilmoitusten keräämisen ja analysoinnin automatisointi

Opinnäytetyö 41 sivua
Huhtikuu 2018

Novatron Oy halusi kehittää ohjelmistojensa vianhallintaprosessia automatisoimalla kaatumisvedosten keräämisen kriittisten ohjelmistovikojen osalta. Tämän avulla haluttiin varmistaa, että kriittiset viat saadaan havaittua mahdollisimman aikaisessa vaiheessa tarpeellisella tarkkuudella, jotta ohjelmiston laatua voidaan parantaa.

Lähdekoodissa esiintyvät virheet voivat aiheuttaa poikkeuksen sovelluksen suorituksen aikana. Poikkeus voi johtua esimerkiksi laittomasta muistiosoiteviittauksesta tai puskurin ylivuotamisesta. Käsittelemättömät poikkeukset johtavat suoritettavan sovelluksen kaatumiseen.

Windows-käyttöjärjestelmä tarjoaa Structured Exception Handling -mekanismin (SEH), jonka avulla C++ -kielellä kehitetyn ohjelmiston poikkeuskäsittelyä voidaan laajentaa hardware-tason poikkeuksiin. SEH:n rajapinnan avulla on mahdollista rekisteröidä sovellukseen oletuspoikkeuskäsittelijä, jota kutsutaan, kun sovellus kohtaa poikkeuksen jolle ei löydy muuta käsittelijää. Käyttöjärjestelmän rajapinnat mahdollistavat kaatumisvedoksen kirjoittamisen poikkeuskäsittelijästä.

Kaatumisvedosten keräämistä varten on kehitetty useita valmiita kirjasto- ja työkalukokoelmia, joiden avulla kaatumisvedokset pystytään myös lähettämään takaisin kehittäjälle ja niiden sisältöä pystytään analysoimaan. Kirjasto- ja työkalukokoelmista Googlen kehittämä Breakpad vaikutti tarkoitukseen parhaiten soveltuvalta valinnalta. Sen etuihin kuuluu alustariippumattomuus ja se, että sovelluksen symbolitaulut voidaan erottaa jaettavasta ohjelmasta.

Kaatumisvedokset tarjoavat tehokkaan keinon sovelluksen viimeisten hetkien tarkasteluun ennen kaatumista. Kaatumishetkeltä kerättävän datan avulla vian syitä voidaan myös tilastoida ja luoda arvio sen kriittisyydestä. Kerätyn datan avulla vika pystytään myös toistamaan mahdollisimman samanlaisessa ympäristössä kuin se on oikeasti esiintynyt. Kaatumisvedos ei kuitenkaan tarjoa kaikenkattavaa näkemystä kaatumisen lopullisista syistä, mutta tarjoaa hyvän lähtökohdan vian syyn selvittämiselle.

Asiasanat: poikkeuskäsittely, C++, Windows, Breakpad, poikkeus, ohjelmisto

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Information and Communication Technology
Software Engineering

KILPELÄINEN VILLE:
Automation of Crash Dump Collection and Analysis

Bachelor's thesis 41 pages
April 2018

Novatron Oy wanted to develop its software debugging process by automating the collection of crash dumps for critical software bugs. This was done to ensure that critical failures could be detected as early as possible with the necessary technical accuracy to help improve the quality of the software.

Bugs in the source code may cause an exception to raise during the execution of a process. An exception may be raised by for example an illegal memory access or buffer overflow. Unhandled exceptions will cause the execution of a process to end.

The Windows operating system provides a Structured Exception Handling (SEH) mechanism that extends the exception handling of software developed in C++. SEH API allows an application to register a top-level exception handler that is executed when unhandled exception is raised. Using Windows API it is possible to write crash dump from the exception handler.

Several ready-made libraries and tool collections have also been developed to enable collection of crash dumps. These libraries and tools also allow the application to send crash dump back to developer for further analysis. From the libraries and tool sets that were examined for this work Google Breakpad was the best suited for the purpose. It is a multiplatform solution and it also allows symbol tables to be stripped from the software before deployment.

Crash dumps provide an effective way to see the last operations executed by the software before the crash. With the ability to collect data from the crashed application the severity of the fault can be estimated. It also allows the developer to reproduce similar environment to examine the bug more closely. However crash dumps don’t necessarily reveal the cause of the bug but it provides a good starting point for debugging the cause.

Key words: exception handling, C++, Windows, Breakpad, exception, software

SISÄLLYS

1 JOHDANTO ... 6
2 LÄHTÖTILANNE ... 7
2.1 Ongelman kuvaus ... 7
2.2 Kehitysympäristö ... 7
2.3 Tavoite ... 8
3 VIANHALLINTA ... 9
4 SYITÄ OHJELMAN KAATUMISEEN ... 11
4.1 Poikkeus ... 11
4.2 Poikkeuskäsittely C++ -kielessä ... 11
4.3 Poikkeusten lähteitä ... 13
5 VIAN ETSIMINEN ... 16
5.1 Staattinen koodianalyysi ... 16
5.2 Profilointityökalut ... 16
5.3 Koodikatselmointi ... 16
5.4 Debuggaus ... 17
6 KAATUMISILMOITUKSET ... 19
6.1 Kaatumisvedosten kerääminen Windows-käyttöjärjestelmässä ... 20
6.2 CrashRPT -kirjasto ... 21
6.3 Google Breakpad -kirjasto ... 22
6.4 DrMingw JIT debugger -työkalu ... 24
7 TOTEUTUS ... 26
7.1 Breakpadin asentaminen ... 26
7.2 Breakpadin alustaminen sovelluksessa ... 28
7.3 Minidump -tiedoston käsittely ... 32
7.4 Breakpadin hyödyntäminen sovelluksessa ... 35
8 JOHTOPÄÄTÖKSET JATKOKEHITYS ... 37
LÄHTEET ... 39

ERITYISSANASTO

Python	Suosittu tulkattava ohjelmointikieli, joka helposti laajennettavissa kattavan kirjastokokoelman ansiosta.
C++	C -kielestä kehitetty käännettävä olio-ohjelmointikieli.
Jenkins	Avoimen lähdekoodin automaatioserveri. Mahdollistaa ohjelmistokehitystyössä toistuvien töiden automatisoinnin.
PDB	Program database -tiedosto. Microsoftin formaatti sovelluksen symbolitaulujen tallentamiseen.
DWARF	Laajasti käytetty standardisoitu formaatti sovelluksen symbolitaulujen tallentamiseen.
GCC	Gnu Compiler Collection. GNU-projektin kääntäjien kokoelma. Sisältää mm. C ja C++ -kääntäjät.
Yksikkötestaus	Lähdekoodin yksittäisten osien, kuten funktioiden testaamista.
Järjstelmätestaus	Kehitettävän järjestelmän kokonaisvaltainen testaaminen. Tarkoituksena testata ohjelmiston ja sen kaikkien ominaisuuksien toiminta.
Käännöslippu	Kääntäjälle annettava parametri, joka muokkaa kääntäjän toimintaa.
Kernel-tila	Tila, jossa suoritettavalla koodilla on rajoittamaton pääsy laitteistoon.
User-tila	Tila jossa sovelluksia suoritettaan suurimman osan ajasta. Tarjoaa suoritettavalle prosessille eristetyn virtuaalimuistialueen, joka suojaa sovellusta aiheuttamasta konflikteja muiden sovellusten kanssa.
Assembly-kieli	Matalan tason ohjelmointikieli, jonka komennot kuvaavat kyseisen arkkitehtuurin konekielisiä komentoja.

1 JOHDANTO

Työn tarkoitus oli kehittää Novatron Oy:n vianhallintaprosessia kehittämällä järjestelmä, joka kerää tietoa ohjelmistossa esiintyneistä kriittisistä vikatilanteista eli kaatumisista ja analysoi niiden syitä. Kehitettävän järjestelmän tulisi havaita ohjelmiston vikatila, kerätä tarvittava tieto kaatumishetkestä ja muodostaa siitä kehittäjille helposti analysoitavaa dataa vian syyn löytämiseksi.

Prosessin automatisoinnilla vika voidaan havaita aiempaa nopeammin ja sen syy saada selville helpommin. Kehitettävän järjestelmän vaatimus oli, että siinä tarvittavat työkalut ja kirjastot olisi saatavilla avoimen lisenssin alaisena ja että se voitaisiin helposti ottaa käyttöön kaikissa Novatronin ohjelmistoprojekteissa.

Työssä perehdytään ohjelmistoissa esiintyvien vikojen perimmäiseen syyhyn ja miten niitä voidaan havainnoida ja välttää. Lisäksi tutustutaan erilaisiin tapoihin kerätä Windows ympäristössä C++ -kielellä kehitetyistä ohjelmistoista kaatumisilmoituksia ja miten niitä voidaan käsitellä ja analysoida vian löytämiseksi.

Lopuksi tutustutaan vielä tarkemmin Googlen kehittämään kirjasto- ja työkalukokoelma Breakpadiin, jota hyödyntäen lopullinen sovellus kehitettiin. Tämän lisäksi pohditaan mitä hyötyä datan keräämisellä saavutetaan ja miten prosessia voisi parantaa vielä entisestään.

2 LÄHTÖTILANNE

Novatron Oy suunnittelee, valmistaa ja toimittaa koneohjausjärjestelmiä maanrakennustyökoneisiin. Koneohjausjärjestelmän avulla kuski pystyy seuraamaan työn etenemistä reaaliaikaisesti työmaasuunnitelman mukaisesti koneen hytistä työnteon ohessa. Tämä tehostaa työskentelyä ja vähentää hukkaa. (Novatron, 2018)

Novatron Oy tarjoaa asiakkailleen ilmaisen etätukipalvelun, joka on käytettävissä suoraan työkoneeseen asennetusta päätelaitteesta. Etätuella on mahdollisuus ottaa yhteys asiakkaan päätelaitteeseen ja nähdä kuskin näkymä laitteesta. Tämä mahdollistaa, että vikatilanteissa tekninen tuki pystyy todentamaan havaitun ongelman ja raportoimaan sen. Tämä onkin aiemmin toiminut pääasiallisena kanavana vikatilanteiden raportointiin.

2.1 Ongelman kuvaus

Novatron Oy halusi kehittää mahdollisuuksiaan ohjelmistossa havaittujen kriittisten vikojen raportointiin ja analysointiin. Tärkeää olisi, että viat voitaisiin havaita mahdollisimman aikaisessa vaiheessa ohjelmistokehitystä ja niistä saataisiin mahdollisimman yksityiskohtaista tietoa, jolloin niihin voitaisiin reagoida tehokkaammin.

Erityisesti haluttiin, että kriittiset viat saadaan korjattua ennen niiden päätymistä julkaisuversioon. Lisäksi halutaan varmistaa, että julkaisuversioiden kriittisten vikojen syyt pystytään selvittämään ja korjaamaan mahdollisimman nopeasti. Tiedonkeruun automaatiolla vikatilanteesta pystytään keräämään tarpeeksi kattava kuvaus viasta ja ympäristöstä, jossa vika on havaittu.

2.2 Kehitysympäristö

Novatron Oy:n ohjelmistokehitys tapahtuu pääasiallisesti C++ -kielellä, jonka kääntäminen tapahtuu MinGW -ympäristössä. MinGW on lyhenne sanoista minimalist Gnu for Windows. Käytännössä se tarjoaa kokonaisvaltaisen avoimen lähdekoodin ohjelmistokehitysalustan natiivien Windows-sovellusten kehittämiseen. Oleellisena osana se tarjoaa porttauksen gcc -käännöstyökaluista Windows-alustalle. (Mingw, 2018) Ohjelmiston lähdekoodia hallinnoidaan versionhallintaohjelmistolla.

Versionhallinnan avulla, lähdekoodiin tehdyistä muutoksista pystytään pitämään kirjaa ja se mahdollistaa useamman kehittäjän työskentelemisen saman projektin parissa. Versionhallinta mahdollistaa myös siirtyminen ohjelmiston eri versioiden välillä vaivattomasti. Lähdekoodin lisäksi myös kehitysympäristön tiedostoja hallinnoidaan versionhallinnalla, jolloin voidaan varmistaa, että kehitysympäristöön tehdyt muutokset, eivät aiheuta ristiriitoja vanhempien ohjelmistoversioiden kanssa.

Ohjelmistokehitys tapahtuu noudattamalla jatkuvan integraation periaatteita, jossa jokainen muutos ohjelmiston lähdekoodissa testataan. Tämän avulla varmistetaan, että ohjelmistosta on aina olemassa ehjä versio versionhallinnassa. Jatkuvan integroinnin työkaluna käytetään Jenkins-automaatioserveriä.

2.3 Tavoite

Työn tarkoitus oli kehittää sellainen ratkaisu, joka olisi yhteensopiva Novatronissa käytettävän kehitysympäristön kanssa ja jonka avulla ohjelmasta voitaisiin kaatumishetkeltä kerätä raportti ja lähettää se edelleen analysoitavaksi. Analyysin on tarkoitus tuottaa kehittäjille sellaista tietoa, joka auttaa vian havaitsemisessa. Ratkaisun tulisi olla sellainen, että se voitaisiin ottaa käyttöön kaikissa Novatronin ohjelmistokehitysprojekteissa vaivattomasti.

3 VIANHALLINTA

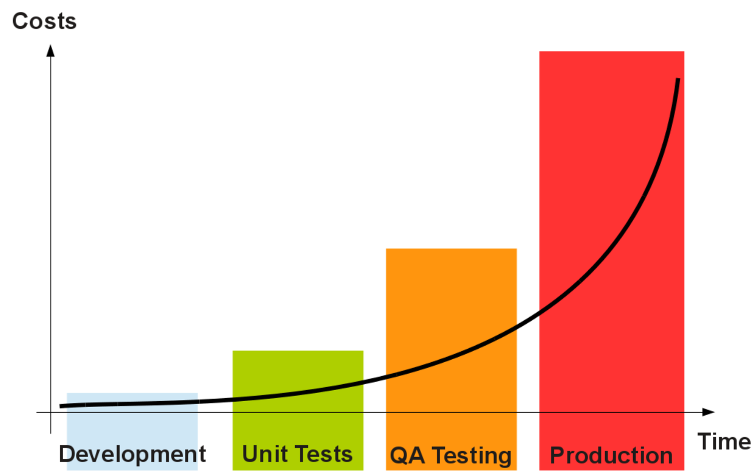
Ohjelmistokehityksessä voidaan olettaa, että valmiista tuotteesta löytyy aina vikoja. Haikala ja Mikkonen esittävät, että valmiissa ohjelmassa voidaan arvioida olevan yksi virhe muutamaa kymmentä riviä kohden ja pitkään käytössä olevassakin ohjelmassa vielä virhe jokaista tuhatta riviä kohden. Heidän mukaan osa virheistä jää myös kokonaan huomaamatta elinkaarensa aikana sillä virhe ei aina aiheuta toimintahäiriötä, mutta pahimmillaan virhe voi johtaa järjestelmän kaatumiseen. (Haikala ja Mikkonen 2011)

Ohjelmistosta voidaan löytää vikoja testaamalla, mutta se ei kuitenkaan takaa ohjelmiston virheettömyyttä. Testauksen tavoitteena onkin paremmin osoittaa ohjelman toimiminen kuin taata ohjelman virheettömyys. (Haikala ja Mikkonen 2011)

Mitä aikaisemmassa ohjelmistokehityksen vaiheessa vika löydetään, sitä halvemmaksi sen korjaaminen yleensä tulee. Jos vika voidaan todeta yksikkötestauksen avulla, on sen korjaaminen usein yksinkertaista, sillä kehittäjällä pitäisi olla kehitettävän ominaisuuden yksityiskohdat edelleen hyvin mielessä, eikä vian korjaaminen edellytä muiden ihmisten työpanosta. (Code Development, 2018)

Vastaavasti vian korjaamisesta aiheutuvat kustannukset kasvavat mitä myöhäisemmässä vaiheessa ohjelmistokehityksen elinkaarta vika havaitaan. Vian korjaamisen kustannukset eivät kasva yksinomaan siitä syystä, että myöhemmässä vaiheessa havaittu bugi olisi vaikeampi korjata. Vian korjaamisen hintaa nostaa se, että myöhemmissä vaiheissa vian korjaamiseen ja testaamiseen joudutaan varaamaan enenevissä määrin henkilöresursseja. Testien suorittamisessa kestää pidempään, jonka lisäksi vian raportointi ja korjaus vaativat useamman työntekijän panoksen. (Code Development, 2018)

Alla olevassa kuvassa (Kuva 1) on havainnollistettu vian korjaamisesta aiheutuvien kustannusten kasvu siirryttäessä elinkaaren vaiheesta toiseen. X-akselilla on kuvattu ohjelmistokehityksen elinkaaren eri vaiheet kehityksestä tuotantoon asti ja y-akselilla kuvattu kustannuksia. Väripalkeilla on esitetty kyseisessä vaiheessa havaitun vian korjaamisen kustannukset ja musta viiva havainnollistaa suhteellista osuutta kustannuksista. (Code Development, 2018)



KUVA 1. Ohjelmistosta löydetyn vian korjaamisesta aiheutuneet kustannukset ohjelmistokehityksen elinkaaren eri vaiheissa. (Software development costs, 2012)

4 SYITÄ OHJELMAN KAATUMISEEN

Tässä luvussa on tarkasteltuna syitä, jotka johtavat ohjelmiston suorituksen ennenaikaiseen loppumiseen. Aluksi tutustutaan poikkeukseen käsitteenä ja miten poikkeuskäsittely hoidetaan C++ -kielessä. Tämän jälkeen tarkastellaan vielä yksityiskohtaisemmin millaiset ohjelmistovirheet voivat johtaa ohjelman kaatumiseen.

4.1 Poikkeus

Poikkeus on äkillinen muutos prosessorin tilassa ohjelman suorituksen aikana, jolloin ohjelman suoritus keskeytetään ja suoritus siirretään poikkeuskäsittelijälle (Randal ym. 2003). Poikkeukset voivat olla lähtöisin raudasta (hardware) tai ohjelmistosta (software) ja ne voivat tulla suoritettavasta sovelluksesta itsestään tai käyttöjärjestelmän ydintoiminnoista. Sovelluksen suoritus voidaan väliaikaisesti siirtää käyttöjärjestelmälle sen rajapintaa hyödyntäen. Käsittelemättömät poikkeukset johtavat suoritettavan ohjelman kaatumiseen. (Exception Handling, 2018)

4.2 Poikkeuskäsittely C++ -kielessä

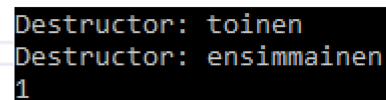
Poikkeus on merkki poikkeuksellisesta tapahtumasta. Ohjelman osa voi aiheuttaa poikkeuksen throw-lauseella, joka aiheuttaa pinon purkamisen, kunnes sopiva käsittelijä löydetään. Catch-lauseella lohko voi ilmaista halunsa käsitellä kyseisen poikkeuksen tai ”…” -määritelmällä halunsa käsitellä kaikki mahdolliset C++ -kielen tyypitetyt poikkeukset. Käsittelemättömät poikkeukset johtavat terminate-funktion kutsuun, joka kaataa suoritettavan ohjelman. (Stroustrup, 2013)

Jos catch-lohko ei osaa palauttaa ohjelmaa poikkeusta edeltäneeseen tilaan, voi ohjelma jäädä tilaan, jossa sen toiminta ei ole enää suunnitellun mukaista. Toisaalta jos ennen poikkeusta varataan resursseja, eikä niitä osata vapauttaa poikkeuskäsittelyn aikana, tämä johtaa muistin vuotamiseen ja voi myöhemmin aiheuttaa ohjelman kaatumisen. Onkin asianmukaista vapauttaa varatut resurssit poikkeuskäsittelyn aikana. Pinon purkamisen aikana kutsutaan pakallisten olioiden hajottimia. Jos resurssit varataan olion rakentamisen aikana ja vapautetaan purkamisen yhteydessä, voidaan varmistua siitä, että resurssit

vapautetaan asianmukaisesti pinon purkamisen aikana. Uudemmissa standardeissa voidaan käyttää hyväksi myös älykkäitä osoittimia, jotka huolehtivat resurssien asianmukaisesta vapautuksesta. (Stroustrup, 2013)

Kuvassa 2 on esitelty C++ -kielen poikkeuskäsittelyn poikkeavaa suoritusjärjestystä ja koodin suorituksen etenemistä. Crash-funktiossa kohdattu throw-lauseke aiheuttaa poikkeuksen, joka aloittaa pinon purkamisen, jolloin ohjelman suoritus siirtyy seuraavaan aaltosulkeeseen. Koska ”toinen” -niminen objekti on paikallinen muuttuja, kutsutaan sen purkajaa ja siirretään suoritus seuraavalle aaltosulkeelle, jolloin myös ”ensimmainen” -niminen objekti tuhotaan. Tämän jälkeen pinon purkamisen aikana löydetään poikkeukselle käsittelijä ja ohjelman suoritus jatkuu siitä normaalisti. Poikkeuskäsittelyn aikana aiheutetaan vielä kuitenkin toinen poikkeus, jolle ei löydy käsittelijää ennen kuin pino on saatu purettua main-funktion tasolle. Tämä aiheuttaa terminate-funktion kutsun, joka kaataa ohjelman. (Stroustrup, 2013)

```
void crash()
{
    Objekti obj("ensimmainen");
    if (1)
    {
        Objekti obj("toinen");
        throw 1;
        std::cout << "Exception thrown" << '\n'; // Not executed
    } // call destructor of "toinen"
} // call destructor of "ensimmainen"

int main()
{
    try {
        crash();
    } // exception caught
    catch (int i)   {
        std::cout << i << '\n';
        throw;
    } // throw not caught call std::terminate()
    return 0;
}
```



KUVA 2. Poikkeuskäsittely ohjelmassa

C++ -standardi ei kuitenkaan tarjoa tapaa käsitellä kaikkia poikkeustilanteita. Esimerkiksi laittomat muistiosoiteviittaukset tai nollalla jakaminen ovat hardware-tason poikkeuksia, joiden käsittely jää käyttöjärjestelmän vastuulle. Oletusarvoisesti käyttöjärjestelmä käsittelee useimmat näistä poikkeuksista lopettamalla ohjelmiston suorittamisen. Kyseisiin virheisiin voidaan kyllä varautua esim. käyttämällä ehtolauseita muuttujien oikeellisuuden tarkastamiseen. (Stroustrup, 2013)

Structured Exception Handling (SEH) on Windowsin keino käsitellä sovelluksen suorituksen aikana kohdattuja poikkeuksia. Sen avulla voidaan käsitellä sekä ohjelmiston että rautatason poikkeuksia. Sen avulla C++ -kielen poikkeuskäsittelyä voidaan laajentaa hardware-poikkeusten osalta Windows-ympäristössä. (Structured Exception Handling, 2018)

4.3 Poikkeusten lähteitä

Kuten edellä todettiin, C++ -kielessä voidaan aiheuttaa poikkeus throw-lausekkeella. Ohjelmointikielen tasolla esiintyvät poikkeukset onkin tarkoitettu käsiteltäviksi kielen tarjoamilla keinoilla. Yleensä sovelluksessa esiintyvät kaatumiset ovatkin virheiden (engl. bug) aiheuttamia. (Kamath, 2013)

Ohjelmistossa esiintyvät virheet voidaan jakaa kolmeen kategoriaan niiden havaitsemiseen liittyvien haasteiden mukaan: muistinhallinnasta johtuviin, samanaikaisuudesta johtuviin ja semanttisiin virheisiin. Ohjelmistossa esiintyvä virhe voi johtaa sovelluksen kaatumiseen, aiheuttaa odottamatonta käytöstä tai aiheuttaa sovelluksen suorituksen näennäisen loppumisen. (Suman, 2013)

Muistinhallinnasta johtuvia virheitä ovat esimerkiksi ylivuodot, muistin vuotaminen, alustamattomien muuttujien lukeminen tai vapautetun resurssin uudelleenvapauttaminen (Suman, 2013). Kuvassa 3 on esitelty viallisesta muistinhallinnasta johtuva virhe. Siinä keosta varattua muistia vapautetaan ensin kutsutun funktion sisällä ja heti uudestaan funktiosta palaamisen jälkeen. C++ -standardin mukaan tämä johtaa määrittelemättömään toimintaan. Se voi esimerkiksi kaataa ohjelman tai korruptoida keosta varattua muistia. (Memory Management, 2018)

```
void crash_ptr(int* ptr)
{
    if (*ptr < 2)
    {
        delete ptr;
    }
}

int main()
{
    int* ptr = new int(1);
    crash_ptr(ptr);

    delete ptr;

    return 0;
}
```

Kuva 3. Esimerkki viallisesta muistinhallinnasta C++ -ohjelmassa

Viallisen muistinhallinnan seurauksena saattaa tapahtua luvattomia muistiosoitelukuja, kun vapautettua resurssia yritetään lukea tai muisti saattaa vuotaa, jolloin lopulta ohjelma ei pysty enää varaamaan uutta muistia. Puskurin ylivuodon seurauksena prosessin aktivaatiotietuepino (luku 5.4) saattaa korruptoitua. (Suman, 2013) Puskurin ylivuotaminen on myös yleisesti tunnettu tietoturvariski (Avoiding Buffer Overruns, 2018)

Semanttisia virheitä ovat kohdat lähdekoodissa, jotka eivät varsinaisesti ole ohjelmointikielen syntaksin näkökulmasta virheellistä, mutta joiden tarkoitus on tilanteeseen nähden väärä. Tällaisten virheiden havaitseminen vaatii lähdekoodin huolellista tarkastelua ja analysointia. Esimerkkinä voisi olla bittitason AND – operaation käyttäminen loogisen AND – operaation sijasta (kuva 4).

```
bool x = true;
bool y = true;

if (check & test)
{
    /*...*/
}
```

Kuva 4. Semanttinen bugi

Samanaikaisuudesta johtuvia virheitä ovat esimerkiksi ns. data race eli tilanne, jossa vähintään kaksi säiettä käyttää samanaikaisesti jaettua resurssia ja vähintään yksi säie myös kirjoittaa resurssiin. Kuvassa 5 on havainnollistettu edellä esitettyä tilannetta. Siinä kolme säiettä t1, t2 ja t3 yrittää kasvattaa saman muuttujan arvoa ilman tahdistusta. Tämän seurauksena muuttuja cnt käyttäytyy hallitsemattomasti.

```
int cnt = 0;
auto f = [&]{cnt++;};
std::thread t1{f}, t2{f}, t3{f}; // undefined behavior
```

Kuva 5. Data Race. (Memory Model, 2018)

Toinen samanaikaisuuden aiheuttama bugi on ns. deadlock, eli tilanne, jossa kaksi säiettä päätyy odottamaan toistensa valmistumista yhtäaikaisesti (Kuva 6). Tämä johtaa sovelluksen suorittamisen näennäiseen loppumiseen, kun kumpikaan säie ei pääse jatkamaan suoritustaan. (Saha, 2013)

```
int main() {
    std::mutex m1;
    std::mutex m2;
    std::thread t1([&m1, &m2] {
        m1.lock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        m2.lock();
    });
    std::thread t2([&m1, &m2] {
        m2.lock();
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        m1.lock();
    });

    t1.join();
    t2.join();
}
```

Kuva 6. Dead lock. Säie t2 lukitsee mutexin m2, jolloin m1 jää lukkoon ja sovelluksen suoritus jumiutuu. (Simple Deadlock Example, 2017)

5 VIAN ETSIMINEN

Ohjelmistokehityksen apuna voidaan käyttää monenlaisia työkaluja, joiden avulla ohjelmistosta voidaan etsiä vikoja. Tässä luvussa on esitelty muutamia yleisimpiä keinoja.

5.1 Staattinen koodianalyysi

Staattisella koodianalyysillä tarkoitetaan ohjelman analysointia suorittamatta koodia. Useimmat modernit kehitysympäristöt tarjoavat sisäänrakennettuina työkaluja, jotka mahdollistavat koodin analysoimisen kehitystyön aikana. Staattisen koodianalyysin ongelma suurissa ohjelmistoissa onkin, että ne tuottavat myös virheellistä informaatiota, joten niiden tuloksiin ei voi varauksetta luottaa. Suodattamalla kehittäjät voivat kuitenkin löytää sieltä oleellisia vikoja. (Penttilä, 2014)

5.2 Profilointityökalut

Koodin suorittamista voidaan analysoida ajonaikaisesti erilaisten työkalujen avulla. Koska C++ -kieli ei hoida itse ns. roskien keruuta, eli vapauta käyttämätöntä muistia, ovat muistinkäytöstä johtuvat virheet tavallisia C++ -kielellä kehitetyissä ohjelmissa. Työkalut analysoivat ohjelman suorittamista ajonaikaisesti ja raportoivat muistinkäytön, samanaikaisuuden ja suorituskyvyn puutteista (Allain, 2018). Suosittuja työkaluja ovat mm. Valgrind ja AddressSanitizer. AddressSanitizer on tarkoitettu erityisesti muistinhallinnan puutteiden profiloimiseen.

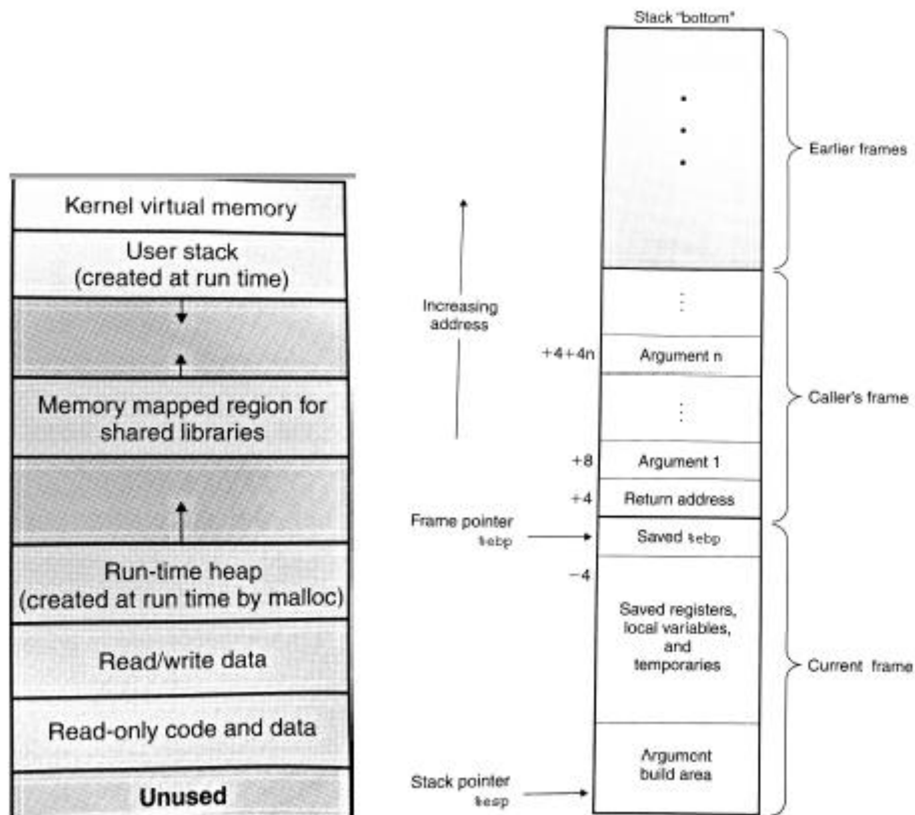
5.3 Koodikatselmointi

Koodikatselmoinnilla tarkoitetaan prosessia, jossa ominaisuuden tekijä hyväksyttää koodinsa ennen kuin se hyväksytään tuotavaksi tuotantoon. Koodikatselmoinnin aikana tiimi käy koodia läpi ja etsii siitä virheitä. Koodikatselmoinnin aikana voidaan myös puuttua seikkoihin, jotka eivät ole varsinaisesti virheitä, kuten tyyliin, helppolukuisuuteen ja suorituskykyyn. Koodikatselmuksia voidaan pitää myös opettavaisina tilanteina, joissa koodaajat saavat paremman ymmärryksen ohjelman toiminnasta ja uudemmat ohjelmoijat oppivat uusia ohjelmointityylejä. (Katselmointi, 2018)

5.4 Debuggaus

Debuggaus on prosessi, jossa ohjelmistossa esiintyvän vian syytä lähdetään etsimään systemaattisesti, kun ohjelman suorituksessa on todettu virhe (Debugging, 2018). Debuggeriksi kutsutaan ohjelmaa, jonka avulla prosessin suoritusta voidaan visualisoida, eli ohjelman etenemistä voidaan seurata rivi riviltä, tarkastella funktiokutsuja ja seurata muuttujien arvoja. Yleensä debuggerit on integroitu osaksi kehitystyökaluja, mutta niitä voidaan ajaa myös itsenäisesti esimerkiksi komentokehotteesta. Debuggeri mahdollistaa myös prosessin virtuaalimuistin tarkastelun. Niiden avulla voidaan esimerkiksi tarkastella sovelluksen aktivaatiotietuepinon sisältöä. (Grötker ym, 2008)

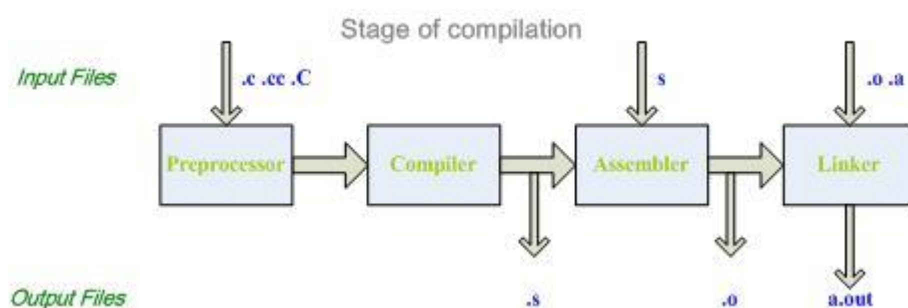
Aktivaatiotietuepino (Kuva 7 oikealla) on muistialue prosessissa (Kuva 7 vasemmalla), jonka avulla ohjelmassa voidaan toteuttaa funktiokutsuja. Funktiota kutsuttaessa aktivaatiotietuepinoon sijoitetaan aktivaatiotietue kyseisestä kutsusta. Se sisältää funktiokutsun parametrit, paluuosoitteen ja tilaa paikallisille muuttujille. (Grötker ym, 2008) Funktion suorittamisen jälkeen kyseinen tietue poistetaan pinosta. (Stack Trace, 2008) Kuvan esimerkit ovat Linux-käyttöjärjestelmästä, mutta niissä esiteltävät periaatteet pitävät paikkansa myös Windows-ympäristössä.



Kuva 7. Prosessin virtuaalimuistin- ja aktivaatiotietuepinon sisältö (Randal ym. 2003)

Ohjelman suorittamisen tarkastelua varten on oleellista, että tarkasteltavasta sovelluksesta on olemassa symbolitaulut. Symbolitaulujen avulla suoritettava ohjelman muistiosoitteet voidaan linkittää lähdekoodissa esiteltyihin muuttujiin ja funktioihin. (Grötker ym, 2008) Onkin oleellista huomata, että jos ohjelman lähdekoodit muuttuvat, symbolitaulutkin pitää päivittää. (TutorialsPoint, 2018)

Symbolitiedot tallennetaan ohjelman käännöksen aikana siitä muodostettavaan binääritiedostoon. Kuvassa 8 on esitelty, miten C++ -ohjelma koostetaan suoritettavaksi ohjelmaksi lähdekoodeista. Käännösprosessi alkaa esiprosessorista. Sen tehtävänä on muokata lähdekooditiedostot siinä määriteltyjen # -merkillä eroteltujen direktiivien perusteella. Tässä vaiheessa esimerkiksi luetaan kaikki include-määreiden sisällöt ja sisällytetään ne varsinaiseen lähdekooditiedostoon. (Randal ym. 2003)



Kuva 8. C++ ohjelman koostaminen. (Basic Introduction gcc, 2007)

Esiprosessilta saapuvat tiedostot ohjataan kääntäjälle, jonka tehtävänä on kääntää lähdekooditiedostot assembly-kielisiksi. Käännöksen jälkeen tiedostot ohjataan assemblerille, joka koostaa assembly-kielisistä tiedostoista edelleen binääriset eli konekieliset versiot. Lopuksi linkkeri yhdistää assemblerilta tulleet objektitiedostot ja ulkopuoliset kirjastoriippuvuudet yhdeksi binääritiedostoksi. (Randal ym, 2003)

Assemblerilta tuleva objektitiedosto sisältää oletusarvoistesti symbolitiedot sen globaaleista muuttujista ja funktioista. Jos ohjelmasta halutaan kattavammat symbolitiedot, voidaan käännös suorittaa antamalla kääntäjälle -g -lippu. Lipun avulla tuotettuja symboleita kutsutaan ns. debug-symboleiksi ja ne kattavat myös moduulin paikalliset muuttujat. Symbolitaulut voidaan poistaa binääristä kokonaan kääntäjälle annettavan -s -lipun avulla. (Randal ym, 2003)

6 KAATUMISILMOITUKSET

Kaatumisvedoksista (engl. crash dump) on ajan saatossa tullut oleellinen osa ohjelmistojen vikojen löytämisessä. Niiden avulla voidaan luoda otos ohjelman tilasta vian sattumisen hetkellä. Kaatumisvedoksen avulla vikaa voidaan analysoida perinteisillä debuggaustyökaluilla. Windows XP:n myötä saapuneen uuden minidump-formaatin ansiosta kaatumisvedoksista on tullut entistä pienempiä kooltaan. Pienemmän kokonsa ansiosta niitä voidaan siirtää tehokkaammin paikasta toiseen. (DebugInfo, 2005)

Minidump-tiedoston sisältämää dataa voidaan muokata käyttämällä erilaisia formaatteja. Tämän avulla voidaan taata, että vian löytämisen kannalta oleellinen data on tallennettu tiedostoon. Listauksessa 1 on esitelty erilaiset minidump-tiedostojen formaatit. (DebugInfo, 2005)

```
MiniDumpNormal
MiniDumpWithDataSegs
MiniDumpWithFullMemory
MiniDumpWithHandleData
MiniDumpFilterMemory
MiniDumpScanMemory
MiniDumpWithUnloadedModules
MiniDumpWithIndirectlyReferencedMemory
MiniDumpFilterModulePaths
MiniDumpWithProcessThreadData
MiniDumpWithPrivateReadWriteMemory
MiniDumpWithoutOptionalData
MiniDumpWithFullMemoryInfo
MiniDumpWithThreadInfo
MiniDumpWithCodeSegs
MiniDumpWithoutManagedState
```

Listaus 1. Minidump formaatit.

Tyypistään riippumatta minidump-tiedosto sisältää järjestelmästä ja suoritettavasta prosessista aina seuraavaa dataa: käyttöjärjestelmän version, prosessorien lukumäärän ja mallin, prosessin tunnuksen, sen luomishetken ja suoritusajat käyttäjätilassa (engl. user mode) ja kernel-tilassa (engl. kernel mode), prosessin lataamien kirjastojen osoitteet, nimet, versiot ja identiteettitiedot, tietoja suoritettavista säikeistä, säikeiden aktivaatiotietuepinot ja tietoja esiintyneestä poikkeuksesta. Kaatumisvedokseen voidaan esimerkiksi sisällyttää prosessin koko virtuaalimuistin sisältö käyttämällä MiniDumpWithFullMemory -formaattia. Tämä vastaavasti kasvattaa vedoksen kokoa. (DebugInfo, 2005)

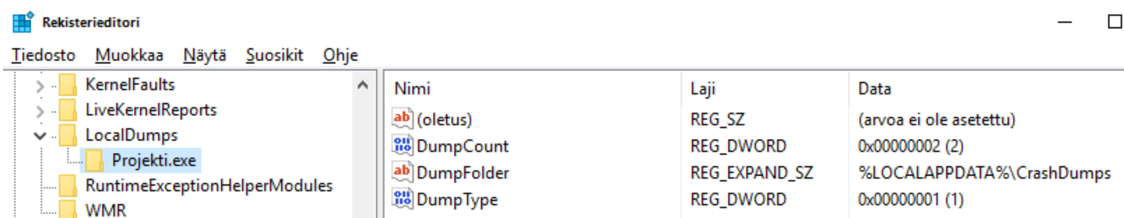
Kaatumisilmoitusten keräämisen automaatiolla voidaan jatkuvasti kerätä tietoa ohjelmistossa esiintyvistä kriittisistä virheistä, jonka avulla ohjelmiston laatua voidaan parantaa. Tietoa keräämällä voidaan myös tilastoida ongelmien vakavuus. Tässä on esitelty erilaisia ratkaisuja kaatumisilmoitusten keräämiseen Windows-ympäristössä.

6.1 Kaatumisvedosten kerääminen Windows-käyttöjärjestelmässä

Windows Vistasta lähtien Windows-käyttöjärjestelmä on sisältänyt Windows Error Reporting -palvelun (WER), jonka avulla sovelluksista voidaan kerätä kaatumisvedoksia muuttamatta sovellusta itseään. Windows Error Reportingia voidaan käyttää kaatumisvedosten keräämiseen ja lähettämiseen Microsoftille, joka tarjoaa sovelluksen käyttäjälle tietoa vian laadusta kerätyn datan perusteella. WER on myös mahdollista konfiguroida siten, että vedoksia kerätään paikallisesti ilman, että niitä lähetetään Microsoftille. Oletusarvoisesti nämä ominaisuudet eivät ole päällä käyttöjärjestelmässä. (Collecting User Mode dumps, 2018)

Windows Error Reporting voidaan asettaa päälle ja sitä voidaan konfiguroida muuttamalla rekisterin arvoja käyttöjärjestelmän rekisterieditorissa avaimella HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps. Jos kaatumisvedoksia halutaan kerätä paikallisesti ilman muuta WER:n tarjoamaa toiminnallisuutta, tulee tämän avaimen alle luoda uusi avain jokaista sovellusta kohden, josta kaatumisvedoksia halutaan kerätä. Uuden avaimen nimeksi tulee laittaa sovelluksen nimi, josta vedoksia halutaan kerätä esim. avain HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Windows Error Reporting\LocalDumps\MyApplication.exe mahdollistaa vedoksien keruun prosessista MyApplication.exe. (Collecting User Mode dumps, 2018).

Kuvassa 9 on esitelty WER:n konfigurointi sovelluksen Projekti.exe osalta. DumpFolder avain-arvo -parin avulla voidaan määrittää, minne kaatumisvedokset tallennetaan, DumpCount:n avulla voidaan määrittää, montako tiedostoa kansio voi korkeintaan sisältää ennen kuin vanhemmat vedokset kirjoitetaan yli. DumpType on puolestaan enumeraatio kaatumisvedoksen formaatille. Esimerkissä (kuva 9) DumpType arvona on käytetty arvoa 1, joka vastaa edellä esiteltyä minidump-formaattia. Rekisterimuutosten jälkeen käyttöjärjestelmä on vielä käynnistettävä uudelleen asetusten tallentamiseksi.

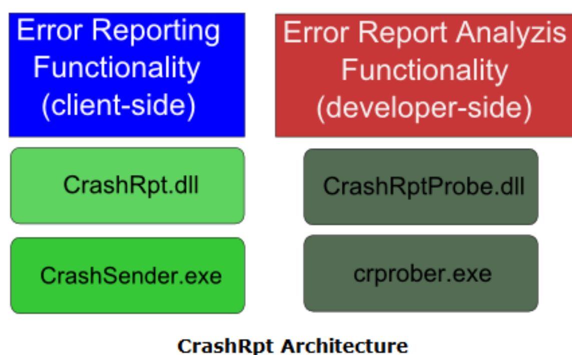


KUVA 9. Windows Error Reporting konfigurointi

6.2 CrashRPT -kirjasto

Kaatumisvedoksien keräämistä ja analysointia varten on myös kehitetty erilaisia kirjastoja ja työkalukokoelmia. Toimintaperiaate näissä kirjastoissa on, että ne mahdollistavat loppusovelluksessa poikkeuskäsittelijän rekisteröimisen, joka kirjoittaa kaatumisvedoksen ennen sovelluksen suorituksen lopettamista. Kirjastojen mukana jaeltavat työkalut mahdollistavat myös vedosten lähettämisen käyttäjältä takaisin sovelluksen kehittäjälle. Kirjastojen etuna on, että pelkän vedoksen kirjoittamisen lisäksi ne tarjoavat työkalut myös vedosten analysointiin. Yleensä työkalut tuottavat lokin, josta näkee kaatuneen säikeen aktivaatiotietuepinon sisällön. Tämä antaa kehittäjälle mahdollisuuden tarkastella sovelluksen suoritusta kaatumista edeltäneeltä hetkeltä, joka helpottaa ohjelmistossa olevan vian etsimisessä. Windows-alustalla C++ -kielelle tarkoitetuista kirjastoista CrashRPT ja Google Breakpad vaikuttivat laadukkaimmilta.

CrashRPT on ilmainen, avoimen lähdekoodin kirjasto ja yhden miehen projekti, jonka avulla sovelluksen kehittäjät voivat kerätä kaatumisilmoituksia sovelluksesta ja lähettää ne takaisin kehittäjälle analysoitavaksi (CrashRPT, 2015). Sen moduulit on kuvattu kuvassa 10.



Kuva 10. CrashRPT arkkitehtuuri (Krivtsov, 2015)

CrashRPT.dll ja CrashSender.exe muodostavat kokonaisuuden, jonka avulla kehitettävästä sovelluksesta voidaan kerätä kaatumisvedoksia ja lähettää ne kehittäjälle analysoitavaksi. CrashRPT.dll -kirjasto tarjoaa rajapinnan, jonka avulla poikkeuskäsittelijä voidaan rekisteröidä sovellukseen. Poikkeuskäsittelijä voidaan konfiguroida myös keräämään kaatumishetkeltä kuvakaappauksia tai jopa videota. Kaatumishetkellä sovellus käynnistää CrashSender -ajotiedoston, jonka avulla kaatumisvedokset ja muu kaatumishetkeltä kerätty aineisto voidaan lähettää takaisin kehittäjälle. Sovellus myös kysyy käyttäjältä lupaa tietojen lähetykseen. (CrashRPT, 2015)

Kun sovelluksen kehittäjä saa käyttäjiltä kaatumisilmoituksia, voi kehittäjä käyttää CrashRPT -kirjaston crprober-työkalua niiden analysoimiseksi. Työkalu on tarkoitettu sovelluksen kehittäjän käytettäväksi, eikä sitä ole tarkoitus jaella kehitettävän sovelluksen mukana, sillä virheraportin analysointi tarvitsee sovelluksen symbolitaulut toimiakseen, eikä niitä ole tarkoitus jaella sovelluksen mukana. Crprober-työkalu osaa purkaa kaatumisvedoksen sisällön ihmisluettavaan muotoon mukaan lukien aktivaatiotietuepinon sisällön. Näiden tietojen perusteella sovelluksen kehittäjän on mahdollista saada selville esiintyneen virheen syy. CrashRptProbe -kirjasto tarjoaa rajapinnan oman analyysityökalun kehittämiselle (CrashRPT, 2015)

CrashRPT -kirjasto käyttää hyödyksi Windows-käyttöjärjestelmän Structured Exception Handler -rajapinnan SetUnhandledExceptionFilter -funktiota, jonka avulla käsittelemättömät poikkeukset voidaan napata. Kun poikkeus tapahtuu, voidaan poikkeuskäsittelijästä kirjoittaa minidump-tiedosto MiniDumpWriteDump -funktion avulla. (CrashRPT, 2015) CrashRPT -kirjasto vaikutti erittäin lupaavalta ja monipuoliselta, mutta lopulta se ei soveltunut käytettäväksi osana kehitettävää järjestelmää. Tämä johtui siitä, että CrashRPT on tarkoitettu käytettäväksi Windows-ympäristössä Visual Studiolla kehitettyjen sovellusten kanssa. Kaatumisvedosten analysointi vaatii symbolitaulut Visual Studion käyttämässä PDB -formaatissa, eikä gcc -kääntäjä pysty tuottamaan symbolitauluja kyseisessä formaatissa. Tämän lisäksi kirjaston kehitys on lopetettu tätä nykyä.

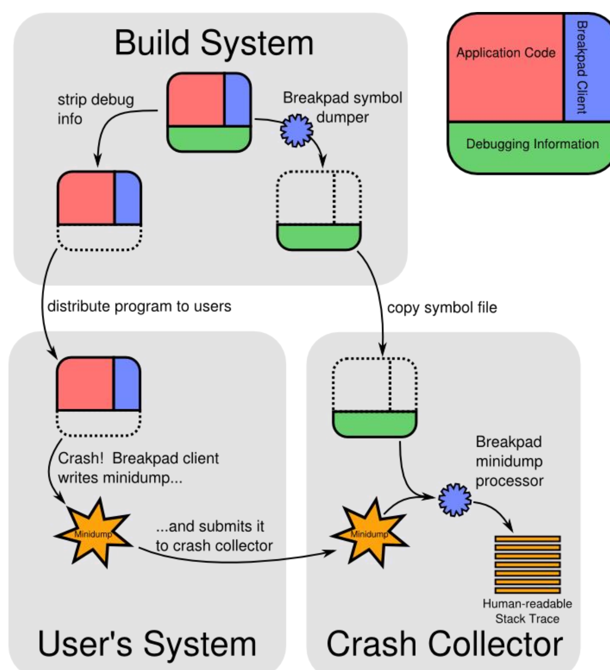
6.3 Google Breakpad -kirjasto

Googlen kehittämä Breakpad on hyvin pitkälti ominaisuuksiltaan ja toiminnoiltaan CrashRPT:ia vastaava kokonaisuus. Se on kokoelma kirjastoja ja työkaluja, joiden avulla

sovelluksesta voidaan kerätä kaatumisvedoksia minidump-formaatissa. Minidump-tiedoston ja ohjelmasta kerättyjen symbolitaulujen avulla Breakpad osaa muodostaa selkokielisen virhelokin sovelluksen aktivaatiotietuepinosta, josta selviää sovelluksen suorituksen vaiheet kaatumiseen johtaneelta hetkeltä. Breakpadin etuihin kuuluu, että se on alustariippumaton, joten sitä voidaan käyttää yhtä hyvin Windows-, Linux- tai jopa macOS-ympäristössä. Monialustaisuudestaan johtuen sitä voidaan käyttää myös monien symbolitauluformaattien kanssa. Näistä syistä lopullinen sovellus päätettiin toteuttaa Breakpadin avulla. (Getting Started With Breakpad, 2017)

Windows-ympäristössä Breakpadin toiminta perustuu CrashRPT:n tavoin Structured Exception Handler -rajapinnan käyttöön ja kaatumisvedoksen kirjoittamiseen. Breakpadin kirjaston tarjoama poikkeuskäsittelijä rekisteröidään loppusovelluksessa ja sovellus osaa kirjoittaa minidump-tiedoston aina kun sovelluksen suorituksen aikana tapahtuu poikkeus, jota sovellus ei osaa käsitellä. (Getting Started With Breakpad, 2017)

Kuvassa 12 on esitelty, miten Breakpadin tarjoamia työkaluja voidaan hyödyntää ohjelmiston vioista raportoimiseen. Käännetyn sovelluksen symbolitaulut tallennetaan Breakpadin tarjoaman dump syms -työkalun avulla, jonka jälkeen sovelluksen symbolitaulut voidaan irrottaa ohjelmasta kokonaan. Symbolitaulujen poistaminen ohjelmasta pienentää jaettavan binäärin kokoa ja vaikeuttaa sovelluksen takaisinmallinnettavuutta. (Getting Started With Breakpad, 2017)



KUVA 12. Breakpadin toimintaperiaate (Getting Started With Breakpad, 2017)

Kaatuessaan sovellus tuottaa minidump-tiedoston. Tiedoston luomisen jälkeen se voidaan lähettää takaisin sovelluksen kehittäjälle analysointia varten. Toisin kuin CrashRPT -kirjastossa, Breakpad jättää sovelluksen kehittäjän vastuulle päätöksen, miten vedos lähetetään analysoitavaksi. Kaatumisvedos voidaan prosessoida tekstimuotoiseksi virhelokiksi Breakpadin tarjoaman minidump stackwalk -työkalun avulla. Työkalu tulkitsee minidump-tiedostoa ja yhdistelee sen sovelluksesta erotettujen symbolitaulutiedostojen kanssa. (Getting Started With Breakpad, 2017) Breakpadin käyttämistä on esitelty tarkemmin luvussa 7.

6.4 DrMingw JIT debugger -työkalu

Ohjelman kaatumisen syitä voidaan analysoida myös ilman kaatumisvedoksen keräämistä. Just-in-time debuggaus on ominaisuus, jonka avulla kaatuneen ohjelman tilaa voidaan tarkastella kaatumisen hetkellä. Sovelluksen kaatuminen aiheuttaa debuggerin käynnistymisen, jolloin sovelluksen tilaa voidaan vielä tarkastella ennen sen suorituksen lopettamista. (Just-In-Time Debugging, 2018)

Dr. MinGW on JIT-debugger -sovellus, mutta sen kirjastot tarjoavat myös rajapinnan, jonka avulla se voidaan integroida osaksi kehitettävää sovellusta. Dr.MinGW pystyy tulkitsemaan symbolitauluja gcc:n tuottamassa dwarf-formaatissa. Se on saatavilla lähdekoodeineen osoitteesta https://github.com/jrfonseca/drmingw. Sivustolta on ladattavissa sovelluksen kirjastojen lähdekoodit ja ohjelman julkaisuversio. Julkaisuversio voidaan asentaa käyttöjärjestelmän oletusdebuggeriksi käynnistämällä se -i -lipulla, jolloin se osaa näyttää sovelluksen kaatuessa kuvan 11 mukaisen virhelokin. Kuvassa on nähtävissä sovelluksen aktivaatiotietuepino aina main-funktiosta (kuvassa alhaalla) virheen aiheuttaneeseen funktiokutsuun asti. (Just-In-Time Debugging, 2018)

```
Dr. Mingw
File  Help
sample.exe caused an Access Violation at location 0000000076D9ECC0 in module msvcrt.dll Writing to location 0000000000000001.

Registers:
eax=00003039 ebx=00000064 ecx=00000001 edx=0028fe00 esi=00003039 edi=0000006f
eip=76d9ecc0 esp=0028fc0c ebp=0028fde0 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010202

AddrPC   Params
76D9ECC0 0028FE00 00404024 00000000  msvcrt.dll!_input_l
76D9EDC8 76D9E991 00404024 00000000  msvcrt.dll!_snwscanf_s_l
76D9ED67 00404027 00404024 00000001  msvcrt.dll!sscanf
0040158E 00000008 60000000 40166666  sample.exe!Function  [Z:\projects\drmingw\sample/sample.cpp @ 10]
     8:
     9: static void Function(int i, double j, const char * pszString) {
>   10:     sscanf("12345", "%i", (int *)1);
    11: }
    12:
0040283E 00000004 40B33333 00000003  sample.exe!StaticMethod  [Z:\projects\drmingw\sample/sample.cpp @ 15]
    13: struct Class {
    14:     static void StaticMethod(int i, float j) {
>   15:         Function(i * 2, j, "Hello");
    16:     }
    17:
0040285E 00401F90 00000000 00000007  sample.exe!Method  [Z:\projects\drmingw\sample/sample.cpp @ 19]
    17:
    18:     void Method(void) {
>   19:         StaticMethod(4, 5.6f);
    20:     }
    21: };
004015A9 00000001 00862BB0 00861D80  sample.exe!main  [Z:\projects\drmingw\sample/sample.cpp @ 25]
    23: int main() {
    24:     Class instance;
>   25:     instance.Method();
    26:     return 0;
    27: }
004013DE 7EFDE000 0028FFD4 779A9F72  sample.exe!__tmainCRTStartup
7660338A 7EFDE000 5275CAC7 00000000  kernel32.dll!BaseThreadInitThunk
779A9F72 004014E0 7EFDE000 00000000  ntdll.dll!__RtlUserThreadStart
779A9F45 004014E0 7EFDE000 00000000  ntdll.dll!_RtlUserThreadStart
```

Kuva 11. Dr. MinGW Kaatumisilmoitus (Dr.MinGW, 2015)

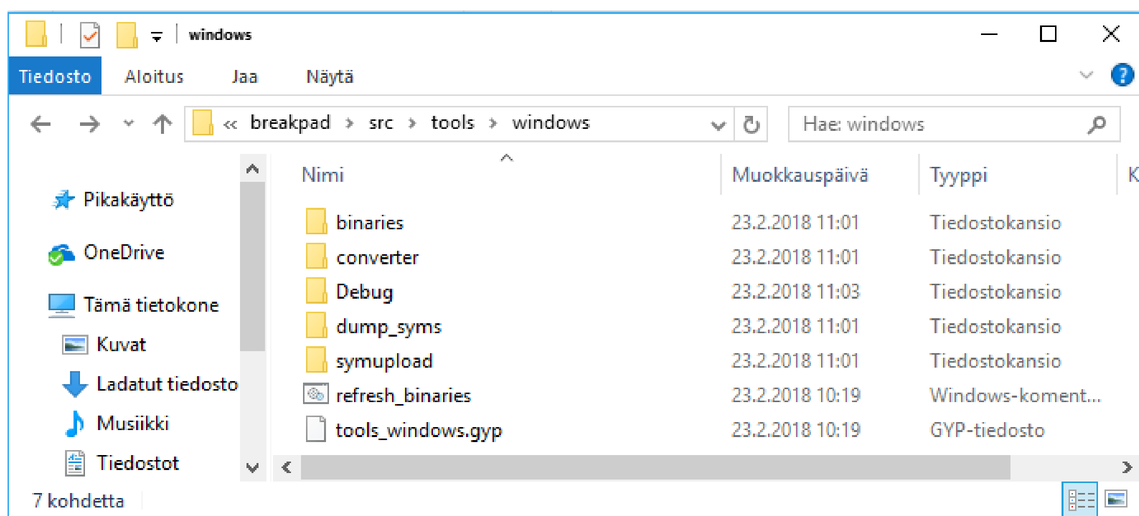
Sovellustasolla Dr. MinGW voidaan ottaa käyttöön linkkaamalla projekti exchnd.dll:n kanssa tai lataamalla se ajonaikaisesti käyttöön. Ohjelmassa pitää vielä kutsua ExcHndInit -funktiota. Dr. MinGW ei tuota kaatuneesta ohjelmasta kaatumisvedosta vaan suoraan virhelokin tekstimuodossa, jolloin suoritettavasta prosessissa tulee olla symbolitaulut tallessa. (Just-In-Time Debugging, 2018) Koska symbolitaulut on hyvä irrottaa jaettavasta sovelluksesta, JIT -debuggerin käyttö ei sovellu käytettäväksi tuotannossa. Toisaalta se tarjoaa kehityksen aikaisesti helpon tavan saada välitön palaute sovelluksen kaatumiseen johtaneista syistä.

7 TOTEUTUS

7.1 Breakpadin asentaminen

Breakpad on avoimen lähdekoodin projekti ja sen voi ladata GitHubista osoitteesta https://github.com/google/breakpad. Sivustolta on ladattavissa Breakpadin symbolien generointityökalun, minidump-tiedostojen käsittelytyökalun ja poikkeuskäsittelijän lähdekoodit. Lähdekoodit on jaoteltu kunkin alustan mukaisesti omiin kansioihin. Sen mukana tulee myös kullekin alustalle projektitiedosto, jonka avulla lähdekoodeista voidaan koostaa poikkeuskäsittelijän kirjasto ja työkalut. Linuxilla Breakpad voidaan koostaa make-työkalun avulla ja Windows käyttäjiä varten löytyy Visual Studio -projektitiedostoja.

Windows-alustalla työkalut tulevat myös valmiiksi käännettyinä ajotiedostoina, jotka löytyvät repositoriosta kansion src/tools/windows/binaries alta (Kuva 13). Jos ajotiedostot haluaa kääntää itse lähdekoodeista, kansion alihakemistosta löytyy tiedosto tools_windows.gyp, jonka avulla voidaan luoda Visual Studio -projektitiedosto, joka voidaan avata ja kääntää Visual Studiolla.



Kuva 13. Breakpad työkalut

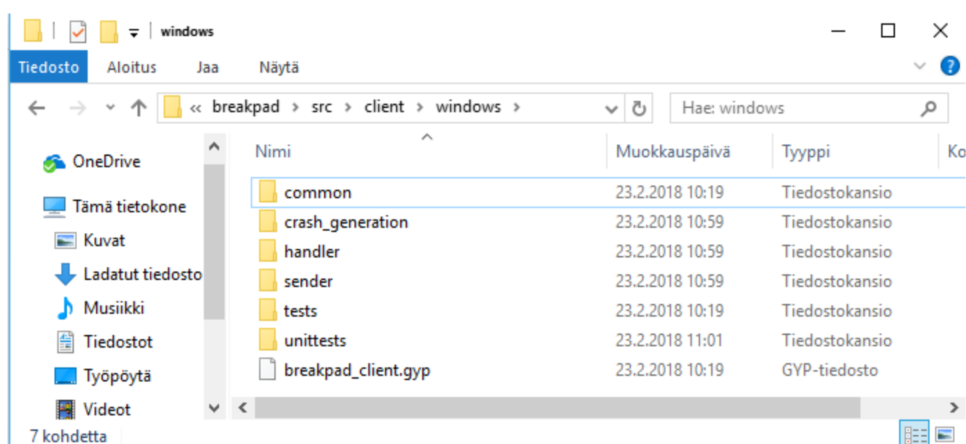
Gyp on googlen kehittämä metakäännöstyökalu, joka voidaan ladata osoitteesta https://chromium.googlesource.com/external/gyp. Sen avulla voidaan muodostaa muita käännöstiedostoja kuten Visual Studio solution -tiedostoja gyp-tiedostossa olevien ohjeiden mukaisesti. Gyp:n asentaminen vaatii Python-tulkin version 2.7. Gyp:n lataamisen ja

asentamisen jälkeen gyp-päätteisiä tiedostoja voi kääntää työkalun juurihakemistosta löytyvän gyp.bat:n avulla. Kuvassa 14 on havainnollistettu Gyp:n asentaminen ja käyttäminen.

```
C:\>cd gyp

C:\gyp>python setup.py install

C:\gyp>gyp.bat c:\breakpad\src\tools\windows\tools_windows.gyp
```

Kuva 14. Gyp:n asentaminen ja gyp-tiedoston ajaminen

Poikkeuskäsittelijän lähdekoodit löytyvät Breakpadin repositoriosta kansiosta src/client/windows (Kuva 15). Sen juuresta löytyy breakpad_client.gyp -tiedosto, jonka avulla sen kirjastoista voidaan muodostaa Visual Studio projektitiedosto edellä mainitulla tavalla. Kun työkalut ja kirjastot on käännetty, on Breakpad valmiina käytettäväksi.



Kuva 15. Google Breakpad kirjaston lähdekoodit

Breakpad on tarkoitettu Windows-ympäristössä käännettäväksi Visual Studiolla. Koska käytössä oli MinGW -kehitysympäristö, edellä esitettyä asennustapaa ei voitu soveltaa. MinGW -ympäristössä Breakpad voidaan asentaa ympäristöön kuuluvan MSYS-työkalun avulla. MSYS tarjoaa Windows alustalle UNIX-tyylisen komentorivitulkin, joka sisältää paketinhallintajärjestelmä Pacmanin. Sen avulla kirjastoja voidaan etsiä ja asentaa komentorivin kautta.

Pacmanin avulla paketteja voidaan etsiä komennolla pacman -Ss nimi (Kuva 16). Paketin asentaminen puolestaan onnistuu komennolla pacman -S paketin nimi. Kuvasta nähdään,

että tässä tapauksessa on asennettuna ylin listattu paketti, joka on 32 -bittiseen ympäristöön sopiva versio.

```
$ pacman -Ss breakpad
mingw32/mingw-w64-i686-breakpad-git r1451.8915f7b-1 [asennettu]
    An open-source multi-platform crash reporting system (mingw-w64)
mingw32/mingw-w64-i686-breakpad-svn r1471-1
    An open-source multi-platform crash reporting system (mingw-w64)
mingw64/mingw-w64-x86_64-breakpad-git r1451.8915f7b-1
    An open-source multi-platform crash reporting system (mingw-w64)
mingw64/mingw-w64-x86_64-breakpad-svn r1471-1
    An open-source multi-platform crash reporting system (mingw-w64)
```

Kuva 16. Breakpad paketit pacman -paketinhallinnassa MSYS-komentorivitulkilla

7.2 Breakpadin alustaminen sovelluksessa

Breakpadin toimintaperiaate on dokumentoitu hyvin sen Github-sivulta löytyvän Documentation-linkin takana. Toisaalta sen asentamisen ja käyttämisen dokumentaatio on osiltaan olematonta tai pahimmillaan harhaanjohtavaa. Parhaimman ymmärryksen sen käytöstä saa lukemalla exception handler -nimisen luokan otsaketiedostoa (Kuva 17).

```
// Creates a new ExceptionHandler instance to handle writing minidumps.
// Before writing a minidump, the optional filter callback will be called.
// Its return value determines whether or not Breakpad should write a
// minidump.  Minidump files will be written to dump_path, and the optional
// callback is called after writing the dump file, as described above.
// handler_types specifies the types of handlers that should be installed.
ExceptionHandler(const wstring& dump_path,
                 FilterCallback filter,
                 MinidumpCallback callback,
                 void* callback_context,
                 int handler_types);

// Creates a new ExceptionHandler instance that can attempt to perform
// out-of-process dump generation if pipe_name is not NULL. If pipe_name is
// NULL, or if out-of-process dump generation registration step fails,
// in-process dump generation will be used. This also allows specifying
// the dump type to generate.
ExceptionHandler(const wstring& dump_path,
                 FilterCallback filter,
                 MinidumpCallback callback,
                 void* callback_context,
                 int handler_types,
                 MINIDUMP_TYPE dump_type,
                 const wchar_t* pipe_name,
                 const CustomClientInfo* custom_info);

// As above, creates a new ExceptionHandler instance to perform
// out-of-process dump generation if the given pipe_handle is not NULL.
ExceptionHandler(const wstring& dump_path,
                 FilterCallback filter,
                 MinidumpCallback callback,
                 void* callback_context,
                 int handler_types,
                 MINIDUMP_TYPE dump_type,
                 HANDLE pipe_handle,
                 const CustomClientInfo* custom_info);

// ExceptionHandler that ENSURES out-of-process dump generation.  Expects a
// crash generation client that is already registered with a crash generation
// server.  Takes ownership of the passed-in crash_generation_client.
//
// Usage example:
//   crash_generation_client = new CrashGenerationClient(..);
//   if (crash_generation_client->Register()) {
//     // Registration with the crash generation server succeeded.
//     // Out-of-process dump generation is guaranteed.
//     g_handler = new ExceptionHandler(.., crash_generation_client, ..);
//     return true;
//   }
ExceptionHandler(const wstring& dump_path,
                 FilterCallback filter,
                 MinidumpCallback callback,
                 void* callback_context,
                 int handler_types,
                 CrashGenerationClient* crash_generation_client);
```

Kuva 17. Exception handler -otsaketiedosto

Breakpadin käyttäminen sovelluksessa vaatii ExceptionHandler -olion luomista mielellään mahdollisimman aikaisessa vaiheessa ohjelman suorittamista, jotta Breakpad voi puuttua esiintyviin poikkeuksiin mahdollisimman aikaisessa vaiheessa sovelluksen suoritusta. Kuvassa 17 on esitelty olion rakentajat, joiden avulla Breakpad voidaan konfiguroida erilaisiin käyttötarkoituksiin.

Kuvassa 17 ensimmäisenä oleva rakentaja on ns. perustapaus, jossa kaatumisilmoitus kirjoitetaan suoritettavasta sovelluksesta kaatumisen yhteydessä. Se ottaa parametreinään polun, johon tiedosto kirjoitetaan, kaksi funktio-osoitinta, context-osoittimen, jolla callback-funktioille voidaan välittää käyttäjän määrittämää tietoa ja handler type -tiedon, jolla voidaan kustomoida Breakpadin poikkeuskäsittelyä.

Loput rakentajat on tarkoitettu pääasiassa sovelluksen ulkopuoliseen poikkeuskäsittelyyn, jolloin minidump-tiedosto kirjoitetaan kaatumisen jälkeen erillisessä sovelluksessa. Erillisen sovelluksen käyttämistä minidump-tiedoston kirjoittamiseen suositellaan varmistamaan, että itse kaatunut sovellus tekee mahdollisimman vähän töitä kaatumisen jälkeen. Sovelluksen ulkopuoliseen prosessointiin käytettäviä rakentajia voidaan kuitenkin käyttää, vaikka minidump kirjoitettaisiin kaatuvasta sovelluksesta itsestään, silloin ylimääräisille parametreille voidaan välittää null-osoitin. Näiden rakentajien avulla voidaan esimerkiksi konfiguroida minidump-tiedoston formaatti MINIDUMP_TYPE -parametrin avulla. Minidump-formaatit on enumeroitu _dbg_common -otsaketiedostossa. Enumeraatiot vastaavat nimeltään luvussa 6.1 käsiteltyjä minidump -formaatteja.

Kuvassa 18 on esitelty funktioprototyypit, jotka määritellään exception handler -otsaketiedostossa. Ne esittävät minkä tyyppinen funktio-osoitin voidaan välittää rakentajalle filter- ja minidump-callbackina. Kirjaston käyttäjän vastuulle jää funktioiden toteuttaminen. Rakentajassa voidaan parametrina antaa myös null-osoitin, jos niitä ei haluta määritellä. FilterCallback -funktiota kutsutaan ennen minidump-tiedoston kirjoittamista, jolloin poikkeusta voidaan tarkastella ja määrittää halutaanko kyseisessä tapauksessa minidump-tiedosto luoda; palauttamalla arvon true se luodaan. MinidumpCallback on puolestaan funktio, jota kutsutaan tiedoston kirjoittamisen jälkeen. Sitä voidaan käyttää tiedoston jälkiprosessointiin. Palauttamalla true, voidaan järjestelmälle ilmoittaa, että poikkeus on saatu käsiteltyä ja sovelluksen suoritus voidaan lopettaa. Jos sovellusta halutaan tutkia jollain muulla poikkeuskäsittelijällä tai debuggerilla, voidaan palauttaa false, jolloin järjestelmä jatkaa poikkeuskäsittelijän etsimistä vaikka minidump olisikin luotu.

```
typedef bool (*FilterCallback)(void* context, EXCEPTION_POINTERS* exinfo,
                               MDRawAssertionInfo* assertion);

typedef bool (*MinidumpCallback)(const wchar_t* dump_path,
                                 const wchar_t* minidump_id,
                                 void* context,
                                 EXCEPTION_POINTERS* exinfo,
                                 MDRawAssertionInfo* assertion,
                                 bool succeeded);
```

Kuva 18. Breakpad-kirjaston callback-funktioiden prototyypit

Handler type on otsaketiedostossa määritelty enumeraatio poikkeusten tyypeille, joihin Breakpadin halutaan puuttuvan (Kuva 19). HANDLER_EXCEPTION määrittelee, että Breakpadin halutaan käsittelevän kaikki poikkeukset, joille ei löydy poikkeuskäsittelijää. HANDLER_INVALID_PARAMETER käsittelee poikkeukset, jotka johtuvat siitä, että

funktioita kutsutaan väärillä parametreilla. HANDLER_PURECALL käsittelee poikkeukset, joissa abstraktin kantaluokan funktioita kutsutaan suoraan. Antamalla rakentajalle parametriksi HANDLER_ALL, voidaan Breakpad asettaa käsittelemään kaikki edellä mainitut poikkeukset.

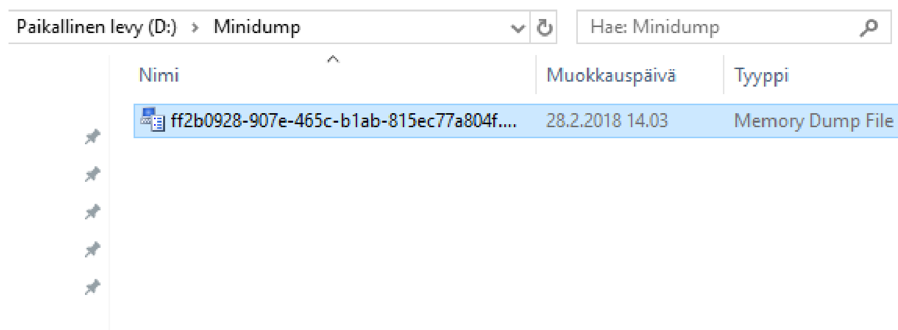
```
// HandlerType specifies which types of handlers should be installed, if
// any.  Use HANDLER_NONE for an ExceptionHandler that remains idle,
// without catching any failures on its own.  This type of handler may
// still be triggered by calling WriteMinidump.  Otherwise, use a
// combination of the other HANDLER_ values, or HANDLER_ALL to install
// all handlers.
enum HandlerType {
  HANDLER_NONE = 0,
  HANDLER_EXCEPTION = 1 << 0,          // SetUnhandledExceptionFilter
  HANDLER_INVALID_PARAMETER = 1 << 1,  // _set_invalid_parameter_handler
  HANDLER_PURECALL = 1 << 2,           // _set_purecall_handler
  HANDLER_ALL = HANDLER_EXCEPTION |
                HANDLER_INVALID_PARAMETER |
                HANDLER_PURECALL
};
```

Kuva 19. Poikkeustyyppien enumeraatio.

Kuvassa 20 on esitelty esimerkki, miten Breakpadin poikkeuskäsittelijä voidaan rekisteröidä käyttöön ohjelmassa. Sen käyttäminen vaatii exception handler -otsaketiedoston sisällyttämistä lähdekoodissa ja sen kirjastojen linkkaamista käännettävään projektiin. Lopuksi ohjelmassa on aiheutettu tarkoituksenmukainen kaatuminen yrittämällä kirjoittaa null-osoittimeen. Tämä aiheuttaa virheellisen muistioperaation ja kuvasta 21 huomataan, että D:\minidump -kansioon on ilmestynyt kaatumisvedos.

```
#include <breakpad/client/windows/handler/exception_handler.h>

bool filterCB(void* context,
              EXCEPTION_POINTERS* exinfo,
              MDRawAssertionInfo* assertion)
{
    return true;
}

bool minidumpCB(const wchar_t* dump_path,
                const wchar_t* minidump_id,
                void* context,
                EXCEPTION_POINTERS* exinfo,
                MDRawAssertionInfo* assertion,
                bool succeeded)
{
    return succeeded;
}

int main()
{
    auto handler = new google_breakpad::ExceptionHandler(L"D:\\Minidump\\",
                                                          filterCB,
                                                          minidumpCB,
                                                          nullptr,
                                                          google_breakpad::ExceptionHandler::HANDLER_ALL,
                                                          MINIDUMP_TYPE::MiniDumpNormal,
                                                          (HANDLE)nullptr,
                                                          (google_breakpad::CustomClientInfo*) nullptr);

    /// Crash here
    int* ptr = nullptr;
    *ptr = 13;

    return 0;
}
```

Kuva 20. Breakpadin poikkeuskäsittelijän alustaminen ohjelmassa



Kuva 21. Ohjelman kaatumisen yhteydessä luotu minidump-tiedosto

7.3 Minidump -tiedoston käsittely

Kaatumisvedoksen luomisen lisäksi Breakpad osaa myös käsitellä ja purkaa tietoa kaatumisvedoksista työkalujensa avulla. Tätä varten suoritettavasta ohjelmasta tarvitaan symbolitaulut. Niiden avulla Breakpad osaa näyttää vian aiheuttaneen rivin lähdekoodista pelkän muistiosoiteviittauksen sijaan. Breakpadin vahvuus on se, että symbolitaulut ja jaettava sovellus voidaan erottaa toisistaan, kun ne on ensin kerätty talteen. Tämä pienentää

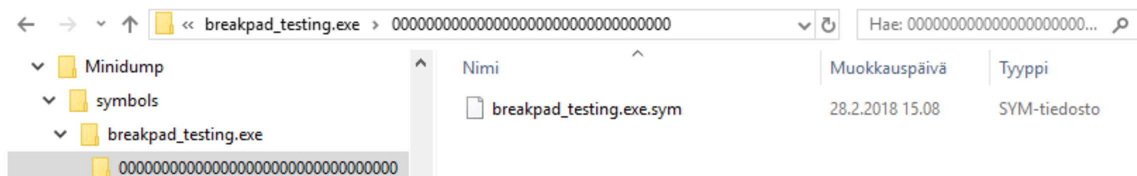
jaettavien binääritiedostojen kokoa ja mahdollistaa sen, että symbolitaulut todella vastaavat jaettavaa sovellusta.

Symbolitaulut voidaan kerätä sovelluksesta talteen Breakpadin dump syms -työkalulla. Sille määritellään tiedosto, josta symbolit kerätään talteen ja sen jälkeen ohjataan tuloste tiedostoon. Tämän jälkeen symboleita voidaan tarkastella vaikka tekstieditorilla, sillä ne ovat ASCII -muotoisia. Kuvassa 22 on esitelty symbolien tuottaminen aiemmin esiteltystä esimerkkisovelluksesta.

```
D:\Koodia\breakpad_testing\cmake-build-debug>dump_syms breakpad_testing.exe > breakpad_testing.exe.sym
```

Kuva 22. Dump syms -työkalun käyttäminen

Jotta Breakpad osaa yhdistää symbolitiedostot minidump-tiedoston kanssa, symbolitiedostot pitää tallentaa oikeanlaiseen hakemistopuuhun. Symbolit tulee tallentaa ”symbols” -nimisen kansion alle, jonka alle luodaan kansio jokaista sovelluksen ajotiedostoa ja kirjastoa varten. Tämän kansion alle luodaan tiedoston versiota vastaava kansio ja vasta tähän kansioon tallennetaan itse symbolitiedosto. Kuvassa 22 on havainnollistettu oikeanlaista hakemistorakennetta.



Kuva 22. Symbolitiedostojen kansiorakenne

Sovelluksen versio voidaan selvittää esimerkiksi tarkastelemalla Breakpadin symbolitiedostoja. Tiedosto alkaa MODULE -tallenteella (Kuva 23), joka kertoo sovelluksen version ja alustan, jolle sovellus on kehitetty. Koska esimerkkisovelluksella ei ole versiota se on kuvassa näkyvä 0:ien jono.

```
MODULE windows x86 000000000000000000000000000000000 breakpad_testing.exe
```

Kuva 23. Symbolitiedoston module-tallenne

Kun symbolitaulut on tallennettu ja asetettu oikeanlaiseen kansiorakenteeseen, voidaan minidump-tiedosto käsitellä minidump stackwalk -työkalun avulla. Se tulkitsee kaatumisvedoksen ja yhdistää sen sovelluksesta kerättyjen symbolitaulujen kanssa. Tämän jälkeen se luo tulosteen, josta näkyy sovelluksen jokaisen säikeen aktivaatiotietuepino siltä osin, kun se on kaatumisvedokseen tallennettu. Tämän lisäksi tulosteessa on vielä tietoa sovelluksen ajoympäristöstä ja sovelluksen lataamista kirjastoista.

Minidump stackwalk -ohjelmalle annetaan parametreina minidump-tiedosto, joka halutaan analysoida ja sovellusta vastaavien symbolitaulutiedostojen juurihakemisto. Tämän jälkeen se tulostaa stderr-virtaan virhetulosteita ja stdout-virtaan kaatumisvedoksen sisällön. Stderr-tuloste on hyvä tapa etsiä syitä, jos Breakpadin tuottama tuloste näyttää vajavaiselta. Breakpadin asettama vaatimus symbolitaulutiedostojen kansiorakenteelle aiheuttaa esimerkiksi helposti virhetilanteen, jossa Breakpad ei löydä sovelluksen tai sen moduulin symbolitauluja.

Minidump stackwalk -työkalu voidaan ajaa esim. alla olevan kuvan mukaisesti (kuva 24). Ylemmässä tavassa sekä stdout- ja stderr-virrat ohjataan tekstitiedostoon. Alemmassa tapauksessa halutaan tallentaa vain kaatumisvedoksen sisältö, joten siinä ohjataan pelkästään stdout-virta tiedostoon virhe.txt.

```
D:\Minidump>minidump_stackwalk ff2b0928-907e-465c-b1ab-815ec77a804f.dmp ./symbols > virhe.txt 2>&1

D:\Minidump>minidump_stackwalk ff2b0928-907e-465c-b1ab-815ec77a804f.dmp ./symbols > virhe.txt
```

Kuva 24. Minidump stackwalk -työkalun käyttäminen

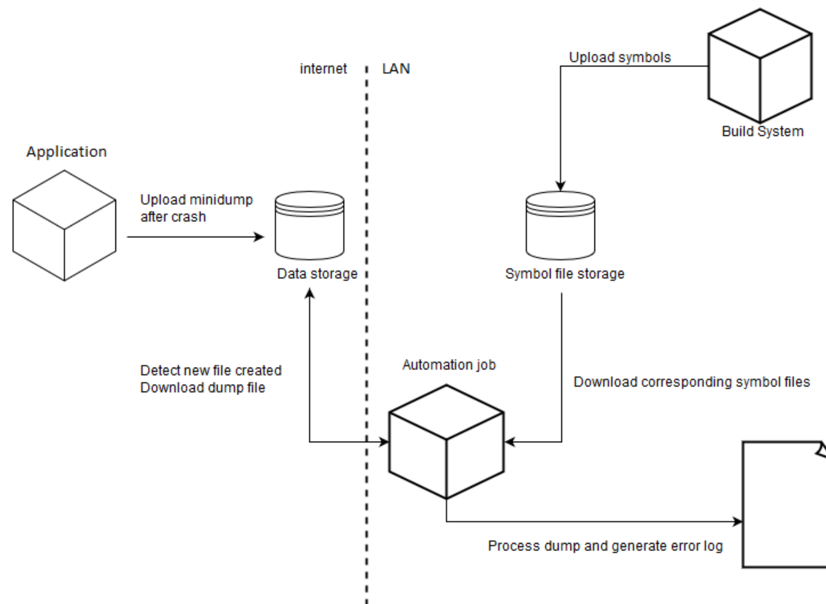
Ajon jälkeen kaatumisen syitä voidaan tutkia katsomalla tuotettua tiedostoa (Kuva 25). Virheloki sisältää tietoa käyttöjärjestelmästä, suorittimesta ja suorituksen aikana kohdatusta poikkeuksesta. Tämän jälkeen tiedostossa on listattuna säikeittäin kunkin säikeen aktivaatiotietuepinon sisältö. Kaatunut säie on aina listassa ylimpänä. Nopeasti huomataankin missä kaatuminen on sattunut, kun luetaan säikeen 0 aktivaatiotietuepinon ylin rivi. Se osoittaa, että kaatuminen on tapahtunut kun main.cpp -tiedoston rivillä 34 ollut koodi on suoritettu. Kun sitä verrataan lähdekoodiin (Kuva 20), huomataan että se on sama kohta, jossa null-osoittimeen koitetaan kirjoittaa arvoa 13.

```
Operating system: Windows NT
                  10.0.16299
CPU: x86
     GenuineIntel family 6 model 94 stepping 3
     8 CPUs

GPU: UNKNOWN

Crash reason:  EXCEPTION_ACCESS_VIOLATION_WRITE
Crash address: 0x0
Process uptime: not available

Thread 0 (crashed)
 0  breakpad_testing.exe!main [main.cpp : 34 + 0x3]
    eip = 0x00401740   esp = 0x006bfe40   ebp = 0x006bfea8   ebx = 0x030272c0
    esi = 0x006bfe6c   edi = 0x00000042   eax = 0x00000000   ecx = 0x006bfe87
    edx = 0x00000001   efl = 0x00010206
    Found by: given as instruction pointer in context
 1  0xf013ee00
    eip = 0xf013ee00   esp = 0x006bfe87   ebp = 0x006bfea8   ebx = 0x030272c0
    esi = 0x006bfe6c   edi = 0x00000042
    Found by: call frame info
 2  breakpad_testing.exe!__tmainCRTStartup [crtexe.c : 334 + 0x1f]
    eip = 0x004013e3   esp = 0x006bfeb0   ebp = 0x006bff80
    Found by: previous frame's frame pointer
 3  kernel32.dll + 0x18654
    eip = 0x76e08654   esp = 0x006bff88   ebp = 0x006bff94   ebx = 0x0ff0019a
    esi = 0x76fc5668   edi = 0x76fc0c30
    Found by: call frame info
 4  ntdll.dll + 0x64a77
    eip = 0x76f24a77   esp = 0x006bff9c   ebp = 0x006bffdc
    Found by: previous frame's frame pointer
 5  ntdll.dll + 0x64a47
    eip = 0x76f24a47   esp = 0x006bffe4   ebp = 0x006bffec
    Found by: previous frame's frame pointer
```

Kuva 25. Breakpadin esitys aktivaatiotietuepinon sisällöstä.

7.4 Breakpadin hyödyntäminen sovelluksessa

Aiemmin esiteltiin Breakpadin poikkeuskäsittelijän ja sen työkalujen käyttämistä yksityiskohtaisesti. Tässä luvussa on kerrottu, kuinka sen avulla on tarkoitus automatisoida kaatumisilmoitusten luominen ja analysointi sovelluksessa. Kuvassa 27 on esitelty kehitettävän järjestelmän arkkitehtuuri yleisellä tasolla.



KUVA 27. Yleisarkkitehtuuri

Ohjelman kääntäminen on automatisoitu Jenkins -automaatioserverin avulla (kuvassa Build System). Kun ohjelman lähdekoodeihin tehdään muutos, Jenkins kääntää ohjelmasta koontiversion ja testaa sen. Kun testit on suoritettu hyväksyttävästi, voidaan sovelluksesta kerätä symbolitaulut talteen. Symbolitaulut kerätään talteen osana Jenkinsin suorittamaa työtä. Se ajaa kääntämisen jälkeen skriptin, joka käy kaikki projektin kirjastot ja ajotiedostot läpi ja kerää symbolit niistä ja asettelee ne Breakpadin kansiorakenteen mukaiseen kansiorakenteeseen. Keräämisen jälkeen tiedostot pakataan ja nimetään koontiversion revision mukaisesti. Revisiotieto mahdollistaa analyysivaiheessa sen, että kaatumisvedos pystytään yhdistämään juuri oikeiden symbolitietojen kanssa.

Sovelluksen kaatuessa, siihen rekisteröity Breakpad-poikkeuskäsittelijä kirjoittaa kaatumisvedoksen ja lähettää sen ulkoverkossa olevalle palvelimelle. Automaatioserverin avulla voidaan tunnistaa palvelimelle saapuneet uudet minidump-tiedostot ja ladata ne sisäverkkoon. Lataamisen jälkeen tiedostoa vastaavat symbolitaulut noudetaan ja kaatumisvedos voidaan prosessoida tekstimuotoiseksi virhelokiksi.

8 JOHTOPÄÄTÖKSET JATKOKEHITYS

Kaatumisilmoitusten kerääminen ohjelmasta tarjoaa helpon tavan saada yksityiskohtaisia kuvauksia ohjelmassa esiintyneestä virheestä. Sen avulla voi nähdä, missä kohdin sovelluksen lähdekoodia on virheen tuottanut osio ja mitä kutsuhierarkiaa on seurattu ohjelman kaatumishetkellä. Kaatumisilmoitus ei kuitenkaan yksinään välttämättä riitä vian perimmäisen syyn löytämiseen, mutta tarjoaa hyvän johtolangan mistä vikaa voidaan lähteä etsimään. Muistinhallintaongelmien havaitsemiseen on lisäksi hyvä käyttää erilaisia työkaluja kehityksen ohessa.

Breakpad tarjoaa tehokkaan keinon kerätä sovelluksesta virheilmoituksia ja käsitellä ne ihmisluettavaan muotoon. Sen etuna on se, että virheloki voidaan muodostaa yhdistämällä sovelluksesta käännöksen aikana erotetut symbolitiedostot kaatumisilmoituksen kanssa. Alustariippumattomuus on myös suuri etu muihin ratkaisuihin verrattuna. Kaatumisilmoitusten keräämiseen löytyi runsaasti vaihtoehtoja, mutta useimmat Windows-alustalle tehdyt ratkaisut oli suunniteltu käytettäväksi Visual Studiolla käännettyjen sovellusten kanssa, jolloin symbolitietojen olisi pitänyt olla sen käyttämässä PDB-formaatissa.

Breakpadin huonona puolena voidaan pitää sen osittain puutteellista dokumentaatiota. Vaikka sen käyttäminen osoittautui lopulta hyvinkin yksinkertaiseksi, sen kääntäminen ja käyttäminen sovelluksessa oli selostettu vaillinaisesti. Pyrinkin tässä työssä avaamaan kattavammin sen käyttämisen yksityiskohtia. Breakpadista tuntui olevan myös kovin vähän keskustelua netin keskustelupalstoilla, vaikka esim. Mozilla käyttää sitä Firefox-selaimessa.

Kehitysympäristö, jolla sovellusta kehitettiin, aiheutti omat ongelmansa. Windows-ympäristölle useimmat kehitetyt työkalut on kehitetty Visual Studion ympärille, joten MinGW-ympäristön käyttäminen tuotti omat haasteensa. Erityisesti ongelmia aiheuttivat symbolitaulujen formaattivaatimukset. Windows-maailmassa oletetaan usein niiden olevan PDB-formaatissa, jota Visual Studion kääntäjä tuottaa.

Kehitettyä järjestelmää olisi tarkoitus jatkossa kehittää eteenpäin. Kaatumisen yhteydessä minidump-tiedostoon kirjoitetaan paljon tietoa myös järjestelmästä, jolla sovellusta on

ajettu. Tämän lisäksi kaatumisen yhteydessä voidaan kerätä muutakin tietoa järjestelmästä, esimerkiksi asetustiedostoja. Datan keräämisellä voidaan varmistaa, että kaatumistilannetta vastaava tilanne pystytään toistamaan mahdollisimman samankaltaisena myös testilaboratoriossa. Sovelluksesta kerättyä tietoa voidaan käyttää hyväksi myös statistiikan keräämiseen, jonka avulla voidaan arvioida vian kriittisyyttä ja esiintymistodennäköisyyttä tietyssä ympäristössä. Jatkossa tulisikin päättää mikä tieto nähdään oleellisena ja miettiä miten se kerätään talteen.

Minidump-tiedostoa pystytään myös käyttämään debuggausprosessissa apuna ja niitä voisi olla hyvä katselmoida osana koodikatselmuksia. Debuggausta varten minidump-tiedosto voidaan avata suoraan esim. WinDBG -työkalulla, jolloin kaatumisvedokseen tallennettua aktivaatiotietuepinoa, voidaan selata jälkikäteen. Aktivaatiotietuepinon selaaminen ja muuttujien arvojen tarkastelu helpottaisi virheen syyn selvittämistä. WinDBG vaatii symbolitiedot PDB-formaatissa, jolloin täytyisi keksiä keino kääntää dwarf-formaatissa olevat symbolitaulut PDB-formaattiin. Kaatumisvedostiedoston käyttäminen debuggausprosessin osana vaatii vielä jatkotutkimista. Kaatumisvedoksista tehtyjen havaintojen perusteella pystyttäisiin havaitsemaan toistuvat virheitä aiheuttavat rakenteet lähdekoodista.

Lopullisena ideana olisi kehittää järjestelmä sellaiseen pisteeseen, että kaatumisilmoitukset prosessoitaisiin ja parsittaisiin tietokantaan, jolle luotaisiin käyttöön web-pohjainen käyttöliittymä. Web-pohjainen ratkaisu mahdollistaisi kehittäjille helpon tavan tutkia ohjelmistossa esiintyneitä vikoja.

LÄHTEET

Allain, A. Using Valgrind to find memory leaks and invalid memory use. Luettu 23.2.2018.
https://www.cprogramming.com/debugging/valgrind.html

Avoiding Buffer Overruns. Luettu 23.2.2018
https://msdn.microsoft.com/en-us/library/windows/desktop/ms717795(v=vs.85).aspx

Basic Introduction gcc. 2007. Luettu 23.2.2018
http://nixchun.pixnet.net/blog/post/12331954-basic-introduction---gcc

Code Development Ltd. 2018. Costs of defect fixing at different stages of development.
http://www.maxtaf.com/press/costs-defect-fixing-different-stages-development

Collecting User Mode Dumps. MSDN. Luettu 23.2.2018
https://msdn.microsoft.com/en-us/library/windows/desktop/bb787181(v=vs.85).aspx

Crash. Wikipedia. Luettu 23.2.2018
https://en.wikipedia.org/wiki/Crash_(computing)

CrashRPT. 2015. Luettu 23.2.2018
http://crashrpt.sourceforge.net/

DebugInfo. 2005. Effective Minidumps. Luettu 23.2.2018
http://www.debuginfo.com/articles/effminidumps.html

Dr.MinGW. 2015. Luettu 23.2.2018
https://github.com/jrfonseca/drmingw/blob/master/img/sample.png

Exception Handling. MSDN. Luettu 23.2.2018
https://msdn.microsoft.com/en-us/library/windows/desktop/ms679329(v=vs.85).aspx

Getting Started With Breakpad, 2017. Luettu 23.2.2018

https://chromium.googlesource.com/breakpad/breakpad/+/master/docs/getting_started_with_breakpad.md

Grötker ym. 2008. Springer. The Developer's Guide to Debugging

Just-In-Time Debugging. MSDN. Luettu 23.2.2018
https://msdn.microsoft.com/en-us/library/5hs4b7a6(v=vs.85).aspx

Krivtsov O. 2015. CrashRPT Architecture overview. Luettu 23.2.2018
http://crashrpt.sourceforge.net/docs/html/architecture_overview.html

Kamath V. 2013. Why do computers crash
https://www.huffingtonpost.com/quora/why-do-computers-crash_b_4226936.html

Katselmointi. Wikipedia. Luettu 23.2.2018
https://fi.wikipedia.org/wiki/Ohjelmistojen_tarkastus

Memory Management. Luettu 23.2.2018
https://isocpp.org/wiki/faq/freestore-mgmt#double-delete-disaster

Memory Model. Luettu 23.2.2018
http://en.cppreference.com/w/cpp/language/memory_model

MinGW. Luettu 23.2.2018
http://www.mingw.org/

Novatron. Verkkosivu. Luettu 23.2.2018
http://novatron.fi/yritys/

Penttilä, E. 2014. Improving C++ Software Quality with Static Code Analysis
https://aaltodoc.aalto.fi/handle/123456789/13471

Randal E, ym. 2003. Pearson Education. Computer Systems – A programmers' perspective

Simple Deadlock Example. 2017. Luettu 23.2.2018
https://gist.github.com/ivcn/227414b3185840434718f7f6c8cbffb1

Software development costs. 2012. Luettu 23.2.2018
https://blog.pdark.de/2012/07/21/software-development-costs-bugfixing/

Stack Trace, Wikipedia. Luettu 23.2.2018
https://en.wikipedia.org/wiki/Stack_trace

Stroustrup, B. 2013. Addison-Wesley Professional. The C++ Programming Language

Structured Exception Handling. MSDN. Luettu 23.2.2018
https://msdn.microsoft.com/library/windows/desktop/ms680657.aspx

Suman, S. 2013. Improving the quality of Error-Handling Code in Systems Software using Function-Local Information
https://hal.inria.fr/tel-00937807

TutorialsPoint, GDB – Debugging Symbols. Luettu 23.2.2018
https://www.tutorialspoint.com/gnu_debugger/gdb_debugging_symbols.htm