Mircea cel Batran" Naval Academy Scientific Bulletin, Volume XX – 2017 – Issue 1
The journal is indexed in: PROQUEST / DOAJ / Crossref / EBSCOhost / INDEX COPERNICUS / DRJI / OAJI / JOURNAL INDEX / I2OR / SCIENCE LIBRARY INDEX / Google Scholar / Academic Keys/ ROAD Open Access / Academic Resources / Scientific Indexing Services / SCIPIO / JIFACTOR

CENTRALIZING APPLICATION CRASH INFORMATION IN SDLC

Laurentiu Alexandru DUMITRU
Eng., Ph.D. (c), Military Technical Academy, 39-49 George Cosbuc Bvd., Bucharest, Romania,
dlaur@nipne.ro

Abstract: *During the testing phase in the Software Development Lifecycle of software applications, implemented safeguards may not catch and treat all possible errors due to various deployment scenarios. When a crash occurs on a client's computer it is more difficult to identify the cause without a proper automated crash reporting framework. Modern operating systems have built-in mechanisms for error reporting but there are also third party cross-platform libraries. With the help of such tools and a centralization system it is possible to implement an efficient problem analysis procedure, when the software runs on a client's computer.*
Keywords: *error reporting, crash dump, SDLC*

I. Introduction

In the current ever growing informational society, software plays a fundamental role on every type of electronic hardware. It has evolved from being distributed off-line to automatic, transparent and even autonomous upgrades and fixes. Although all phases in the Software Development Life-cycle (SDL) process are important, a critical part is represented by the actual running on client machines, which, typically is among the last phases. Every remote client has a particular environment which can have different Operating Systems (OS) and patches, different anti-viruses or firewalls and any other technical peculiarities.

During the implementation phase, the software is usually designed with unexpected error and exception handling. When the software undergoes evaluation and testing, during the Quality Assurance and testing phase, many running scenarios and execution environments are used to spot bugs in the application. Given the vast degree of heterogeneity on remote clients it is almost impossible to imagine and test all the possible situations in which the client application may run.

When an application crashes on a remote client machine, it is very important for the developers to have a sustainable method of retrieving enough information about the crash, in order to track and fix the bug that caused it. When applications were ran on systems that have no Internet connectivity, many years ago, this was a daunting task to accomplish. In present times, most clients have Internet access, at least periodically, if not permanently. This makes it easy to transfer crash-related information over a secure channel from a remote client to a central collecting service.

This paper analyzes how an automated crash reporting service can be developed, what tools are available to software developers in order to implement automatic error reporting on modern operating systems and how should the remote clients access the error collecting framework. The actual bug tracking process is OS and programming language specific, but the collection process can be applied to most scenarios.

Given the fact that after a critical error, the application, usually, is no longer active, crash detection must be implemented with the aid of an external component. Most operating systems provide native application programming interfaces to such services but there are also third party libraries which can be used. Taking into account an automated crash reporting process from the beginning of the development leads to easier problem fixing and better customer support for the final product.

II. Related work

In [1], Kirk Glerum et. al, talk about debugging applications that are distributed to tens of millions of clients, with emphasis on collecting with the Windows Error Reporting (WER) technologies. Apart from describing how the technology works, the paper also describes how the Microsoft team manages data collection, sorting, user response and statistical data.

Following the WER's model, several other projects exists. Apple provides the CrashReporter facility [2], Google provides its multi-platform system – BreakPad [3].

Other reporting systems have been in used [4],[5], but many of them were replaced either with native tools or with alternatives that have larger support communities.

User anonymity is always an important issues. Castro et al. [6] propose a method for using trigger-generating variables instead of memory dumps. This approach is more processor intensive on the client machine. Instead of

DOI: 10.21279/1454-864X-17-I1-064
© 2017. This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 4.0 License.

providing an actual dump, it will provide a way to reproduce that particular event on an analysis machine where the memory contents are filled with arbitrary data, instead of disclosing data from the client's machine

Another method for anonymizing user data is proposed by Scrash [7]. It is a technique that excludes sensitive memory address from the dumps. It works by labeling sensitive information at source code level with the cost of a small execution overhead due to the fact that extra symbols are inserted.

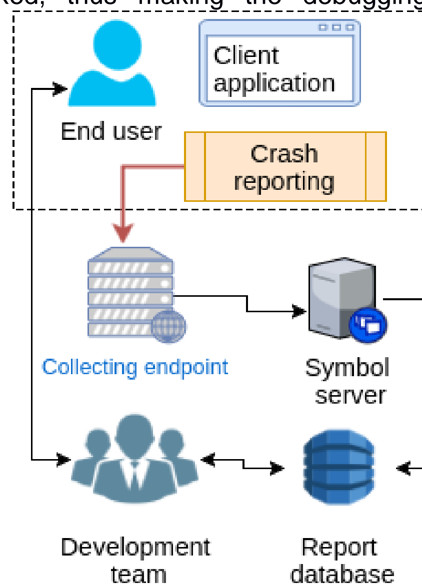
Apart from the technical data collection, the bug report itself is an important asset to the debugging team. In [8] N. Bettenburg et. al., analyze withing the users of Apache, Eclipse and Mozilla what information is needed by the developers and what they actually get from the users.

III. Overall design

The crash collecting architecture can be based on a client-server model. In this case, the client is the crash reporting service running on a remote client machine and the server is the entry-point of the error-collecting infrastructure. Given the fact that a crash report may include a user's confidential data, in the memory dump, it is necessary to have an encrypted communication channel over which the collected information is transmitted. The actual implementation varies. Windows Error Reporting submits information only for Microsoft products and developers that use its services must deploy their own transfer method. On the other hand, third party frameworks may directly provide this facility.

A major architectural decision is whether to have anonymous data submitted or to include user specific information that may link a certain crash dump to a particular user. The common approach is to have various environment informations collected, such as OS version, kernel version or installed/running applications, which might help in the debugging process but can not be used to identify a particular user. If, for example, the user gives its consent to be identified, the debugging team may directly communicate with the user in order to notify him about the status of the problem and even submit particular patches for exceptional situations.

Typically a crash takes place when certain conditions are met. These may include invalid processor instructions, attempt to access a memory address that is not allocated to the current process, attempt to read or write an address that represents a hardware device, attempt to write past the allocated space of a buffer or triggering unhandled exceptions. If the application is not a part of the operating system, the crash may affect only the user's activity. If the application is a part of the operating system, runs with administrative privileges or interacts with OS services, a crash may lead to unrecoverable system errors, kernel panics or system hangs. In these latter cases an error recovery service may not have time to be executed, since the system is blocked, thus making the debugging process



much harder. Modern operating systems have built-in error collecting mechanisms for kernel faults and, usually, normal application will not trigger faults in the kernel.

Current multi-tasking operating systems run each user application in separate virtual address spaces. This approach allows isolation in case of a single application crash. After the detection of the crash, the system can launch a registered error collecting helper. It is this helper that will collect the necessary information and submit it to the collecting server (Fig.1). If the application is designed with personalized support, the helper may identify itself and then submit the information. After the information has been submitted, the user receives a confirmation e-mail that the situation has been acknowledged and he/she will be contacted with regards to the evolution of the bug-fixing process. If the information is submitted anonymously, the users will be announced when a new version of the application is available for download.

Figure 1

The collecting infrastructure would, minimally, include an error collecting server, an application debug symbol database, a dump database and a bug tracking service. The

DOI: 10.21279/1454-864X-17-I1-064
© 2017. This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 4.0 License.

collecting server receives the dumps which are initially stored in the dump database along with user identification, if provided. The next step is identifying the associated symbol files with the received files. The dump is then corroborated with the symbol files in order to have a report that can be processed by a member of the debugging team. Usually this report contains stack frame information, thread state information, CPU registers but can also actual memory contents, if the helper was configured in that way. In most cases, actual memory contents are not included because they are not relevant to crash analysis, have large sizes and may contain user sensitive information.

IV. Implementation

On Microsoft Windows based system, the Windows Error Reporting service is the native tool that can help catch and collect information when an application crashed. According to Microsoft [1], fixing 20 percent of the top-reported bugs can solve 80 percent of customer issues and addressing 1 percent of the bugs would address 50 percent of the customer issues. These numbers show how important proactive bug-fixing can be, in order to have good customer support.

Apple provides the Crash Report Service [2] which allows the developers to download crash reports from application that were distributed and installed on client machines through the store. Submitting the collected crashes is conditioned by the confirmation of the user to share crash data with the application's development team. One feature is the TestFlight beta testing with which the final clients can download beta versions of the applications before they are released. This implies automatic error reporting.

On Linux systems, the Automated Bug Reporting Tool [9], which is natively included on Fedora and Red Hat-derived distributions, catches core dumps from user applications and sends the reports to bug-tracking systems.

When developing cross-platform applications, having multiple collecting methods and crash dump formats can prove to be inefficient. In such a case, the development team may want to implement a custom collecting service or use a readily available one, such as Google Breakpad [3]. This approach assures cross-platform homogeneity of the crash dumps and a lighted storage and collecting service.

Crash dumps must have a well defined format and record as many information as possible. Several crash dump formats exist but their interpretation differs, therefore are not compatible. Google's Breakpad has chosen the Minidump format, which is endorsed by Microsoft's error reporting service. Dump must contain at least CPU context, thread information (memory regions, context for each thread), the Process Environment Block (PEB) for the process and a list of external code that was present at the time the application crashed. In a normal situation, a dump is the result of a crash, but one can trigger a manual dump in order to catch a snapshot of the application context at a certain point in time.

Each minidump contains a series of streams and each stream contains a particular type of data that can be used for analysis. Minidumps can be of many types. The most common is the "normal" one which contains thread and module information, CPU context and other system information. The "full" type contains all the memory segments that were in use by the process. The latter one implies collecting sensitive information. Since the format can be regarded as a container for several types of stream, extending the format to accommodate customization is a straightforward task. The main minidump structure is defined in DbgHelp.h (windows):

```
typedef struct _MINIDUMP_HEADER {
ULONG32 Signature;
ULONG32 Version;
ULONG32 NumberOfStreams;
RVA StreamDirectoryRva;
ULONG32 CheckSum;
union {
ULONG32 Reserved;
ULONG32 TimeDateStamp;
};
ULONG64 Flags;
} MINIDUMP_HEADER, *PMINIDUMP_HEADER;
```

The base RVA of the minidump directory. The directory is an array of MINIDUMP_DIRECTORY structure:

```
typedef struct _MINIDUMP_DIRECTORY {
ULONG32 StreamType;
MINIDUMP_LOCATION_DESCRIPTOR Location;
} MINIDUMP_DIRECTORY;
typedef struct
_MINIDUMP_LOCATION_DESCRIPTOR {
ULONG32 DataSize;
RVA Rva;
} MINIDUMP_LOCATION_DESCRIPTOR;
```

The structure itself does not specify any source code or function reference. A specialized parser corroborates the information from the dump with the symbol files, after matching a particular application version with the appropriate symbol file. Additional symbol files for external components such as DLLs or shared libraries may be loaded if needed. Analysis of minidump files is done with Microsoft Windows Debugger (WinDbg) and can start with a simple *!analyze -v* command,

DOI: 10.21279/1454-864X-17-I1-064
© 2017. This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 4.0 License.

in order to get detailed information about the dump.

The Windows Error Reporting API provides many functions that can create custom reports, filter conditions and submit crash dumps. A handler DLL must be registered in the system registry (HKLM SOFTWARE\ Microsoft\ Windows\ Windows Error Reporting) before it can be used. The function responsible for notifying the system that it should call an external helper in case of a crash is defined as:

HRESULT WINAPI
WerRegisterRuntimeExceptionModule (
_In_ PCWSTR pwszOutOfProcessCallbackDll,
_In_opt_ PVOID pContext);

and must be called before the application starts its actual work. The full path of the registered DLL must be provided as first argument and an arbitrary information can be passed as the second one. The DLL must implement three callback entrypoints:

1. *OutOfProcessExceptionEventCallback*
2. *OutOfProcessExceptionEventSignatureCallback*
3. *OutOfProcessExceptionEventDebuggerLaunch Callback*

The first one is used to determine whether the helper confirms that it processes the crash or not. It is used to filter out known crash conditions (such as manually snapshots) and to gather more information about the situation. The second function can be called by WER multiple times in order to get report parameters. The third function is used to launch a particular debugger with its specific parameters, but it is not necessary to actually launch one.

To get the minidump, it is necessary to call the specific function, inside the second callback of the handler DLL:

HANDLE f = CreateFileW (Path,
GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, 0, NULL);
MiniDumpWriteDump(pExceptionInformation->hProcess, GetProcessId(pExceptionInformation->hProcess), f, MiniDumpNormal,NULL, NULL, NULL);

This example will write a normal minidump to a specified file. After its completion it may be sent to the collecting service.

Using Breakpad is similar to the above procedure. An application must be linked against the breakpad library, provide a callback and register the exception handler:

static bool dumpCallback (const google_breakpad :: MinidumpDescriptor& descriptor, void context, bool succeeded) {*
//send file
return succeeded;
}
google_breakpad::MinidumpDescriptor descriptor("/opt/dumps");
google_breakpad::ExceptionHandler eh(descriptor, NULL, dumpCallback, NULL, true, -1);

The above code instructs the library to use a certain directory for dumps, registers the *dumpCallback* function to be called after a dump is generated and calls the above registered function. Inside the callback an upload procedure must be present in order to submit the dump to the centralized error collecting infrastructure.

Conclusions

Good customer support and small bug-fixing times leads to better applications. Having a proactive error collecting policy can help the development team to quickly identify and repair software flaws. In many cases this is done without user interaction and can help minimize a problem's impact. If, for example, a single problem is reported and treated quickly, the same problem will not manifest on future customer's machines or on present ones that have not stumbled upon the crash conditions. This automated collecting process alongside with an automated update mechanisms should be included in the Software Development Life-cycle Process from the start.

Modern operating systems already provide built-in tools for such features, but external libraries that implement such functionalities exist under open-source and commercial licenses. These libraries are usually well-suited for cross-platform deployments when crash dump format homogeneity is needed. Confidentiality of a client's data is achieved by collecting non-sensitive technical information and by transmitting the collected information over secure transmission channels with encrypted data. Furthermore, the reports are submitted anonymously. There are exceptions in which sensitive information or personal identification is used, but these cases require special circumstances and the user's approval.

The error collecting infrastructure, apart from the crash-related information storage, can be used to provide user feedback and generate reports about bug-fixing times, process or module-related statistics with regards to crash occurrences, thus enabling the development team to identify flawed algorithms, human coding errors and resource/performance-related issues. With this information, future application versions can be improved.

DOI: 10.21279/1454-864X-17-I1-064
© 2017. This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 4.0 License.

Bibliography

[1] Glerum, Kirk, et al. "Debugging in the (very) large: ten years of implementation and experience." *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.
[2] Apple Inc., CrashReporter. Technical Report TN2123, Cupertino, CA, 2004.
[3] Google Inc. Breakpad. Mountain View, CA, 2007, http://code.google.com/p/google-breakpad/.
[4] Berkman, J. Bug Buddy. Pittsburgh, PA, 1999, http://directory.fsf.org/project/bugbuddy/.
[5] Mozilla Foundation. Talkback. Mountain View, CA, 2003, http://talkback.mozilla.org.
[6] Castro, M., Costa, M. and Martin, J.-P. Better Bug Reporting With Better Privacy. In Proc.of the 13th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, pp. 319-328, Seattle, WA, 2008.
[7] Broadwell, P., Harren, M. and Sastry, N. Scrash: A System for Generating Secure Crash Information. In Proc.of the 12th USENIX Security Symposium, pp. 273-284, Washington, DC, 2003.
[8] Bettenburg, Nicolas, et al. "What makes a good bug report?." *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008.
[9] Zdenek Prikryl Moskovcak, Automatic Bug Reporting Tool,https://fedoraproject.org/wiki/Features/ABRT

DOI: 10.21279/1454-864X-17-I1-064
© 2017. This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 4.0 License.