
Game Developers Conference®
February 28 - March 4, 2011
Moscone Center, San Francisco
www.GDConf.com
GDC
UBM

Forensic Debugging

Getting the truth by inference, autopsy, and putting the squeeze on informants

Elan Ruskin
Valve Corporation
GDC 2011

slides available at:
bit.ly/hPCmVW

Good Afternoon!

slides available at:
bit.ly/hPCmVW

Good morning everyone! Please turn off cell phones, fill out eval forms, etc.

Who this talk is for:

- CS grads
  - (who know C++ and how a call stack works)
- Gameplay Programmers
- System Programmers
- ~~Artists~~
  - (try room 303?)

slides available at:
bit.ly/hPCmVW

This talk is for programmers of all levels:

Beginners will learn why the debugger sometimes lies to them and how to beat the truth out of it

Intermediates will learn techniques for dealing with core dumps of all kinds and divining the causes of really weird problems

Experts may be interested by our system for aggregating stability data during testing and even after release, and how we act upon that in support

QA will learn all they need to know about what "crash dumps" are and how to collect them in a way that gets them fixed fast

Artists are in the wrong room.

Today's Show

What is a crash?

Basic Anatomy

The lying debugger

Typical Fingerprints

Collecting Evidence

Data After Shipping

Today's program, in six parts.

First, what is a crash? What exactly happens in the app and the operating system when one is encountered?
Second, some basic anatomy of crash dumps and what they contain.
Third, why the debugger is so often unreliable when dealing with crash data or optimized executables.
Fourth, typical patterns of common crashes, and how to identify them.
Fifth, how to collect all your evidence in one place, and deal with it efficiently.
Sixth, how we collect data and act on stability data from our customers even after we've shipped to them.

This talk is about:

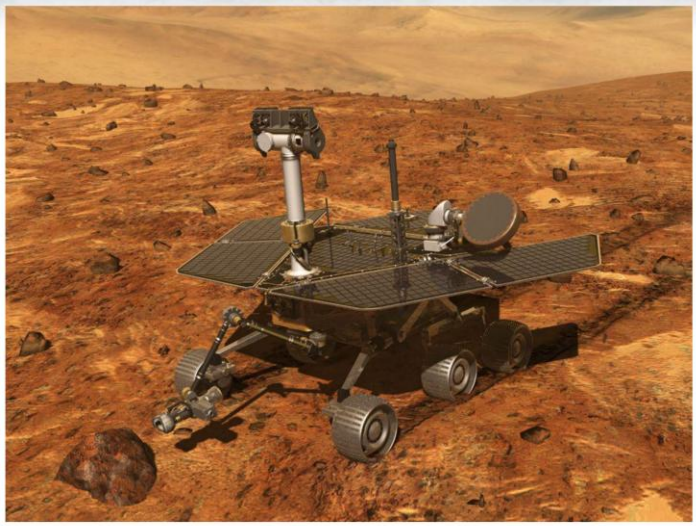
- Crashes
  - Collecting crashes
  - Testing crashes
  - Analyzing crashes
  - Common causes of crashes
  - Fixing crashes
  - A crash course
- The mysterious "Disassembly" window in your debugger and why it is your best friend
- Debugging "release" builds generally
  - How to do it, why to do it, and why not to be afraid
  - How to read a call stack by eye when the tools fail you
  - Tips about MSVC and GDB you may not have known
  - Techniques for a forensic approach to debugging
- Tools for collecting stability data from a lot of machines
  - Best practices for QA while you're still testing internally
  - How to get data from your game after you've shipped it
  - How Valve gets data from five million testers customers around the world and turns it into patches
    - And some handy open source tools: breakpad, socorro, and friends
- Every platform Valve supports:
  - PC, Mac, 360, PS3, Linux

ONCE UPON A TIME,
ON A PLANET FAR FAR AWAY…

Let's start with a story.

The Crash on Mars

A story about the crash on Mars. Not the probe that slammed into it because it mixed up miles and kilometers…



The Little Robot That Couldn't

… but the Spirit rover, which went out of commission for many days because its control computer crashed.

Mars Spirit Rover

- IBM RAD6000 CPU (PowerPC derivative)
- 128mb DRAM
- 256mb Flash card (DOS filesystem)
- VxWorks OS in ROM
- Two radio antennas
  - Low-gain UHF
    - to Earth or to orbiter
    - <1kbps
    - omnidirectional
  - High-gain X-Band
    - 11kbps to Earth
    - 128kbps to orbiter
    - Needs aiming
- 20 minute ping time!

Here's the specs on the Mars spirit rover's computer system. It has two ways of communicating by radio: a low-gain antenna, which is used for emergencies and during landing, and a high-gain antenna, which is like a satellite dish that communicates at a much higher baud. Either antenna can be used to transmit directly to Earth, or (at a faster rate) to the Mars Global Surveyor orbiting overhead, which can relay data between Earth and the probe.

This information from a great NASA paper:
The Mars Rover Spirit FLASH Anomaly
Glenn Reeves
Tracy Neilson
Jet Propulsion Laboratory (JPL)
Pasadena, CA 91109
818-393-1051
Glenn.E.Reeves@jpl.nasa.gov, Tracy.A.Neilson@jpl.nasa.gov

Spirit Timeline

Sol 18

- Roll onto surface
- 9am uplink fails
- 12:45pm “BEEP!” ok
- Other downloads missed

Sol 19

- Spirit uploads empty packet to orbiter
- Scheduled downloads missed
- Spirit sends 7.8125bps “major error” beep via UHF.
- NASA transmits: “send us your status”

Sol 20

- Spirit transmits 5min of telemetry, over and over
- “I’ve rebooted lots of times”
- Battery temperature high, charge low

Sol 21

- Day 21, “low power fault” signal received
- Dedicated backup circuit detects batteries running low, tries to shut down
  - When shutdown fails, backup backup circuit unplugs the CPU!

Key Facts

- Stuck in reboot loop
- Not shutting down overnight to conserve battery
- Not recording telemetry or science to FLASH

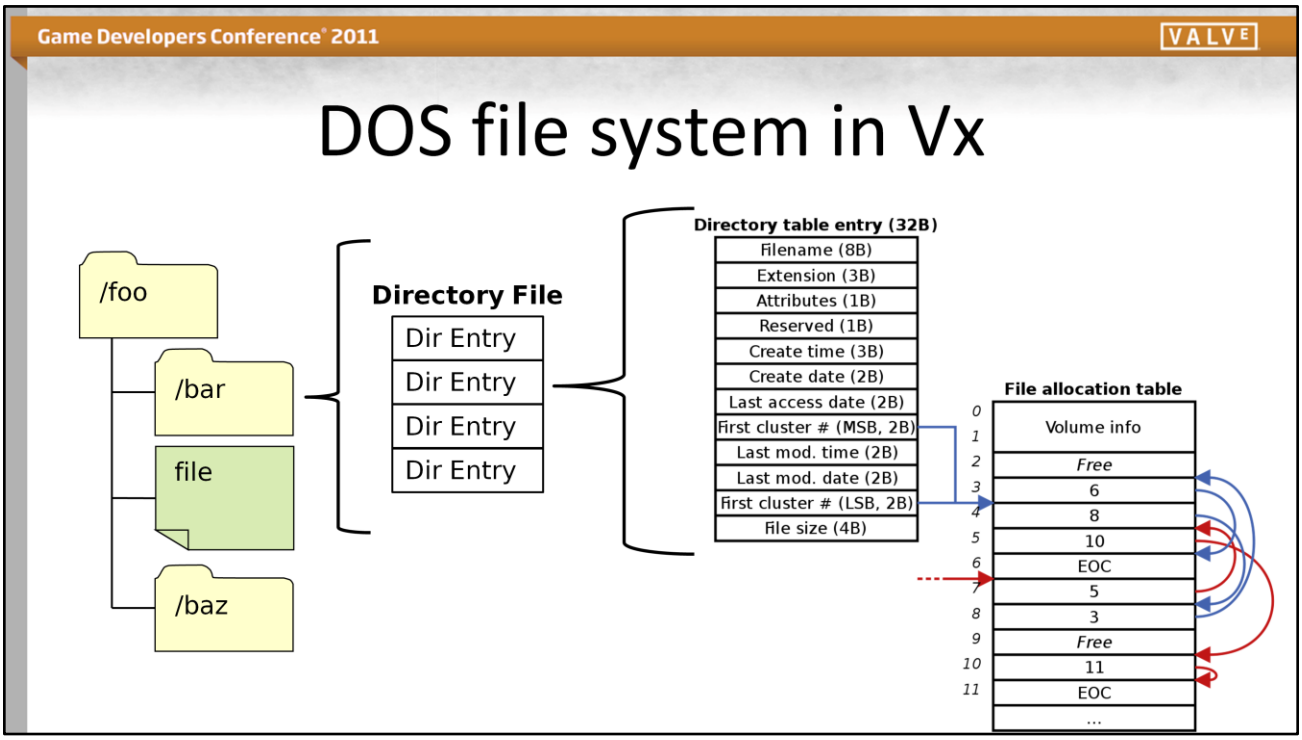
After midnight, the probe successfully turned on its UHF radio and trasnmitted to the orbiter for 2 minutes and 20 seconds, but sent no data. Ie, its radio was on and working properly, but had not been given data to transmit.

The probe missed its next appointed downloads, and finally sent the Blue Bleep Of Death at its “double backup emergency” super low bitrate.

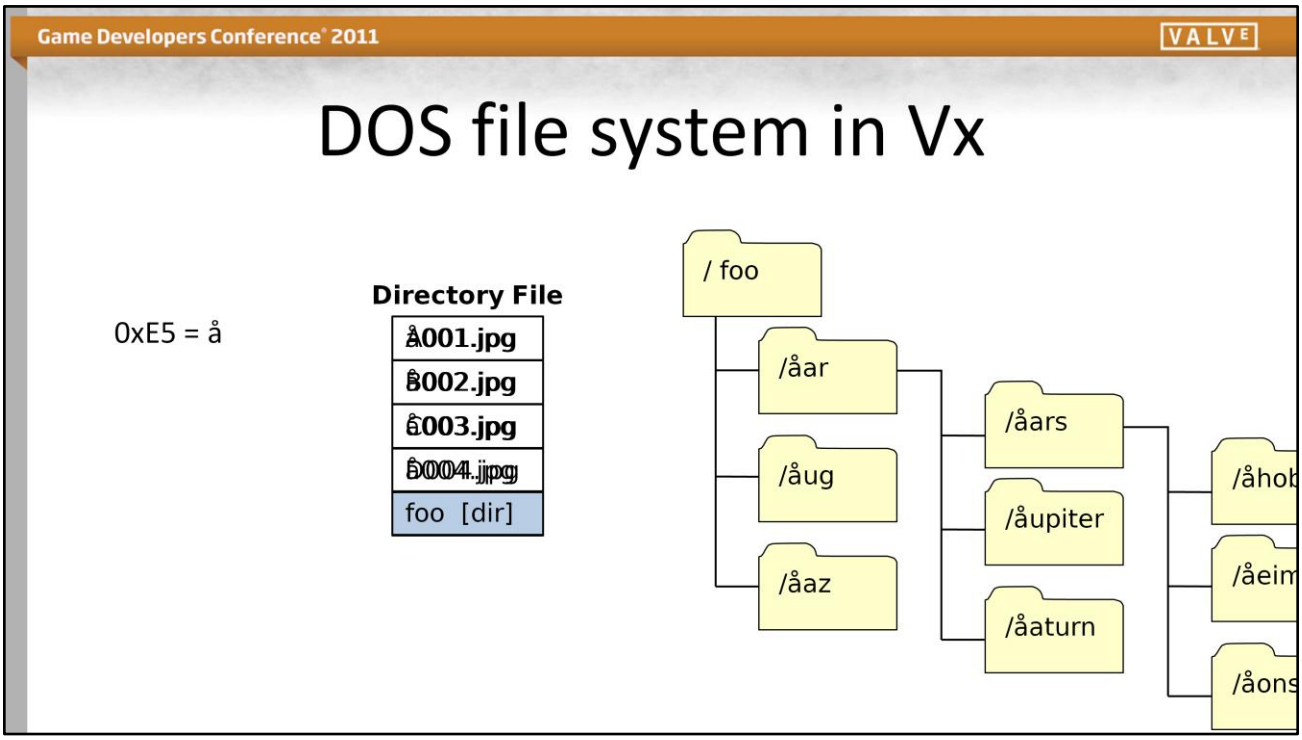
NASA knew four things could put the probe in that mode: low power, broken system clock, loss of uplink, or an X-band radio fault. They knew the clock wasn’t broken because Spirit beeped on time, the “uplink lost” timer hadn’t expired yet, there was no indication of low power on day 18, and an X-band fault wouldn’t have affected the UHF radio.

Workaround

- NASA spams: “go to crippled mode, disable FLASH”
- Batteries recharge over the next day and CPU reboots in crippled mode.
- On day 32, sent “reformat FLASH drive” command
  - fixed problem (temporarily).
  - Why?



The DOS filesystem is a tree of subdirectories.
Each subdir is a file containing a list of filenames and disk addresses
Deleting a file means just overwriting the first byte of its name with 0xE5
Mounting the filesystem means loading the directory trees into RAM



But this means that the amount of RAM needed to mount the filesystem increases with the number of deleted files! Even though the space is available “on disk”, you still need storage to represent all those deleted filenames.

DOS file system in Vx

- Two fatal misconfigurations:
  - DOS library allowed to malloc() space for deleted filenames, and run out of room
  - malloc() deadlocked when out of space
- PowerPC watchdog timer (DEC register) detects deadlocks and triggers exception when they occur (in this case reboot)
- CPU would boot, mount filesystem, run out of room, deadlock, reboot, mount filesystem, run out of room, deadlock….
- NASA uploaded a patch in April 2004

Two configuration errors conspired to place the system into a condition where it would reset repeatedly and also prevented the vehicle from autonomously shutting itself off to save power. A configuration error in the DOS Library module allowed the size of the private memory area to expand by allocating additional space from the free system space12. A configuration error in the Mem Library module silently resulted in a suspended task when the request for additional memory could not be satisfied.

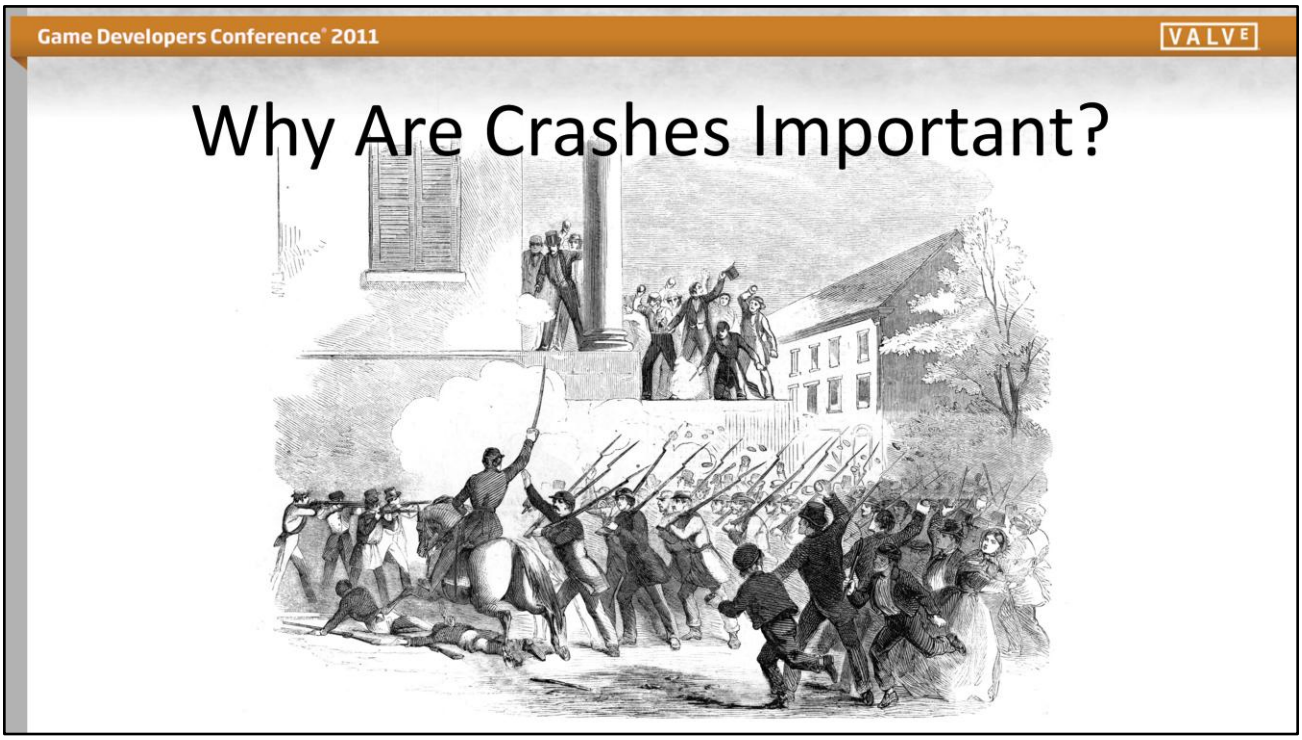
NASA LIVE

…in what is probably the first ever example of interplanetary DLC. The patch included:

Compaction of subdirectories after files are deleted to yield back the used space
Automatically entering crippled mode after repeated resets
Using watchdog timer to force overnight shutdown if “normal” shutdown deadlocked
Put a timeout on the semaphore that waited for the FLASH library functions to return

Oh, and not using malloc after initialization!

And so they Saved Science.



Why do we care about crashes?

Raise your hand if you've ever had a game crash on you.
Keep your hand up if you thought this was really annoying.
Keep your hand up if you ever lost progress to a crash.
Look around the room.
Those people with their hands up are your customers.
[display image]

Even if you hate your customers, an unstable game will usually fail console certification.

The question isn't really "why are crashes important" but "why is it important to fix them this way?"

Why Treat Crashes Specially?

- Extremely annoying to customer
  - The ultimate "showstopper"
- Relatively easy to fix (compared to severity!)
- State snapshot is better than waiting for repro
- A special case of "release mode forensics."

The question is really, why treat crashes differently from other bugs?
Well, they're extremely annoying to the customer, and they can make you fail cert outright more easily than almost any other bug.
But at the same time, once you know how to diagnose them, it's often obvious what caused the crash.
So crashes have a pretty high ratio of how annoying they are over how hard they are to fix.
Also, crashes are unique in ways that makes them debugabble by means not available to most other bugs: in particular, it's often easier *not* to wait for a repro, since you have a snapshot of the crash's state.
And really this talk isn't so much about crashes particularly as a forensic approach to debugging: how you can look at the current state of a process, at its memory and stack, and figure out what's gone wrong and what to do about it.

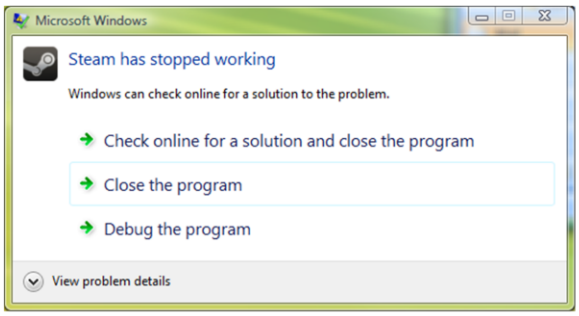
WHAT IS A CRASH?

And why are they hard to debug?

Unless you're in aerospace, when you say "the computer crashed" you don't mean it actually blew up and died. You mean that your application stopped running because it hit some exceptional condition and went to "plan B."

A Crash is "Plan B"

- A programmed response to an exceptional condition.
- Protect system from malfunctioning apps
- Allow recovery
- Provide diagnostics
  - The "core dump"



A crash is a programmed response to an exceptional condition. It's a Plan B you've filed with the operating system ahead of time, to tell it what to do in the case your program can't continue.

Plan B is designed to:

Protect the computer from a malfunctioning program

Allow the user to recover and regain control of their OS (more so in Win/Mac than consoles)

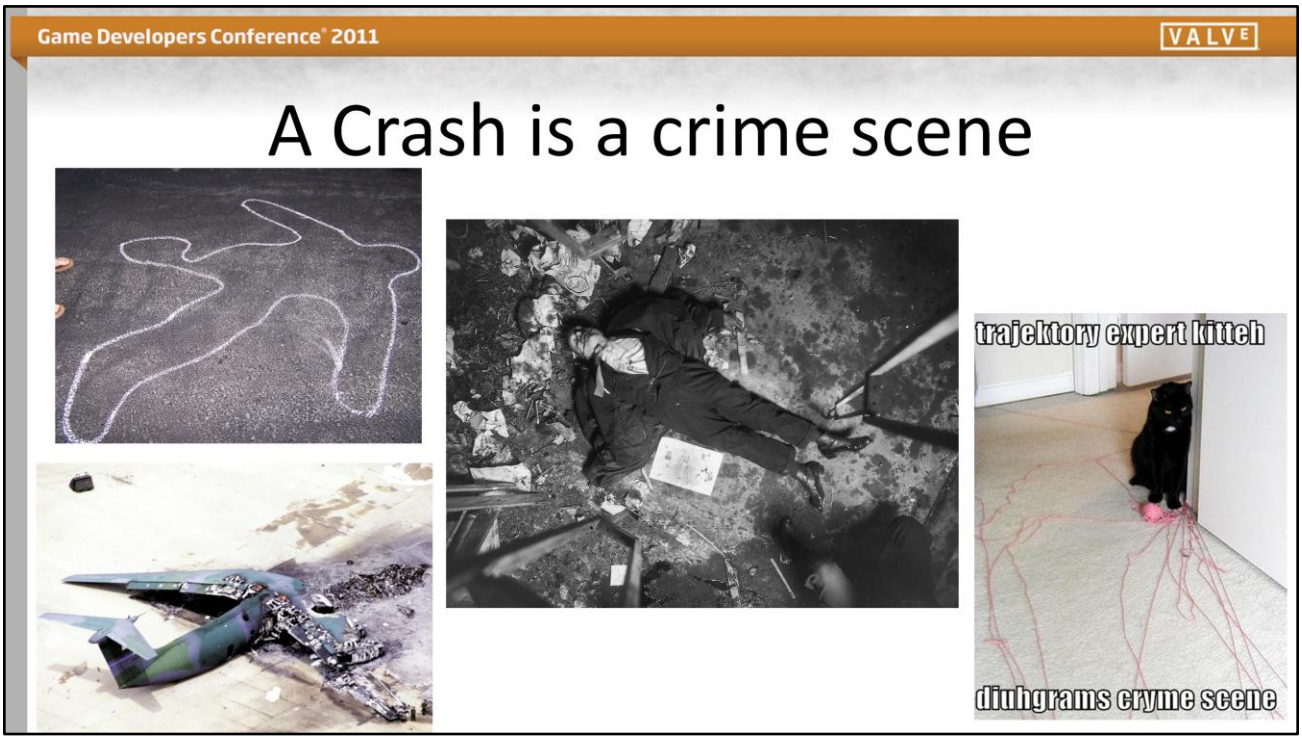
Provide information to help fix the problem

But sometimes this information is hard to interpret

Debugging a crash is like performing an autopsy: you're looking at the state of the thing after it has died, and trying to figure out the sequence of events that got it there

The default exception handler for the Windows OS is this familiar dialog, which lets you close the program, send back diagnostic information, or open a debugger.

This is the technical description, but I find there's another metaphorical way to look at crashes that's a more helpful mindset for actually fixing them.

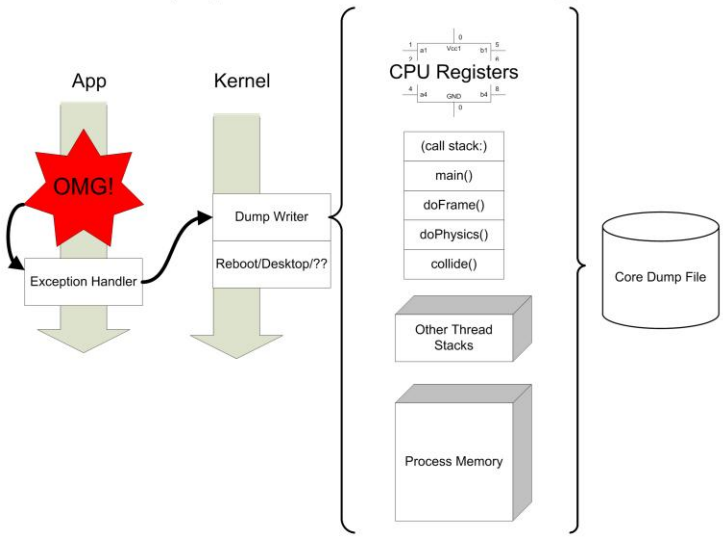


… a crash is a crime scene.

Debugging a crash is like performing an autopsy: you're looking at the state of the thing after it has died, and trying to figure out the sequence of events that got it there

- You have to think like CSI.
- Sometimes the primary cause is in the past and you infer it from secondary evidence
- Sometimes the state of the victim is a little bit ground-up and you need to piece it back together like a puzzle.

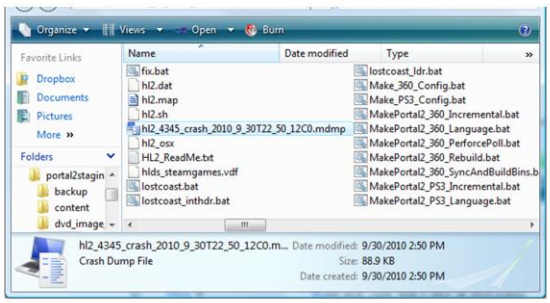
What happens during a crash?

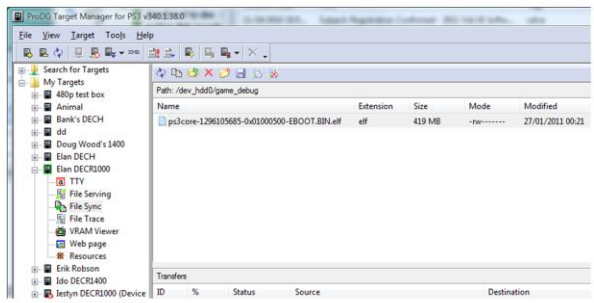


Whatever the cause, the program went into an exception handler -- not a C++ exception, but a "Plan B filed in case of emergencies“
Typically, this halts the program, records some information about the current state of the program, and goes into a recovery state -- shutting down the process in a Windows machine, or simply halting/rebooting a console.
"Recording the current state" can take many different forms, but usually it includes the current state of the CPU, a call stack, and sometimes some of the memory of the current thread stack, other threads, or the process heap.

Where does the dump go?

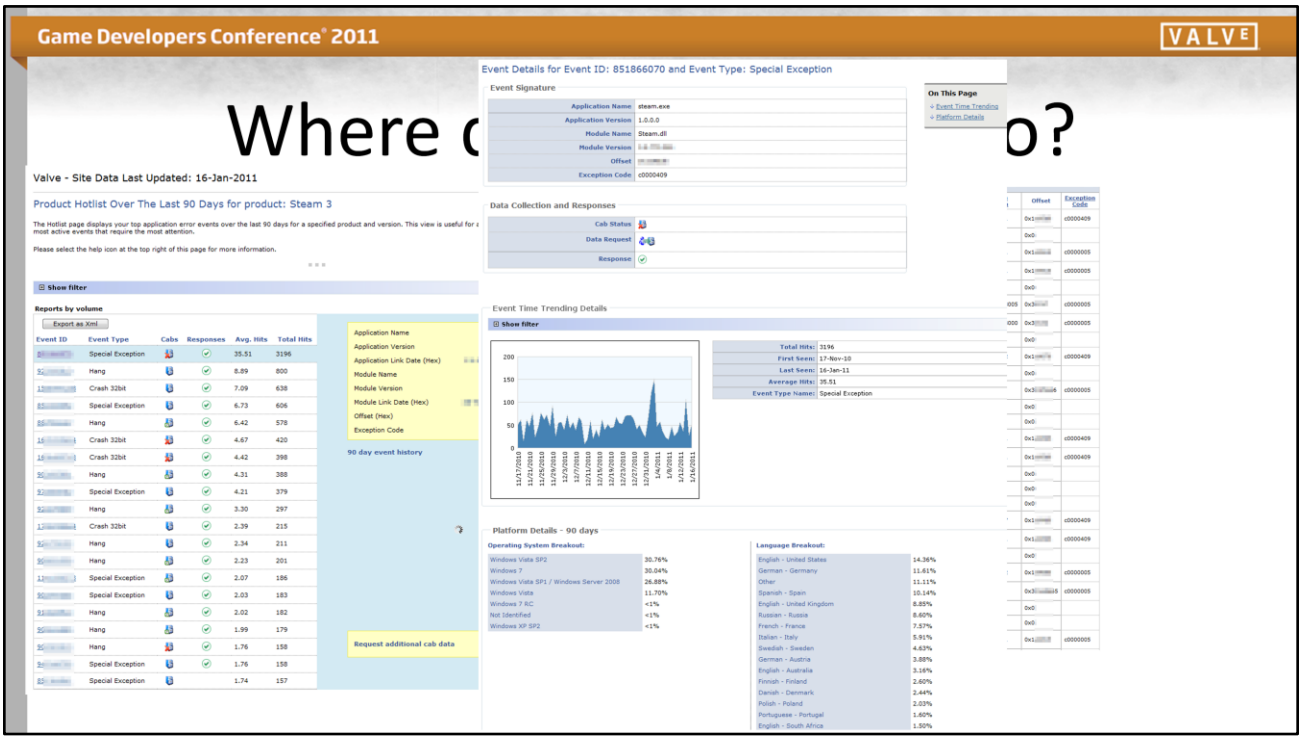
- Dump to Disk



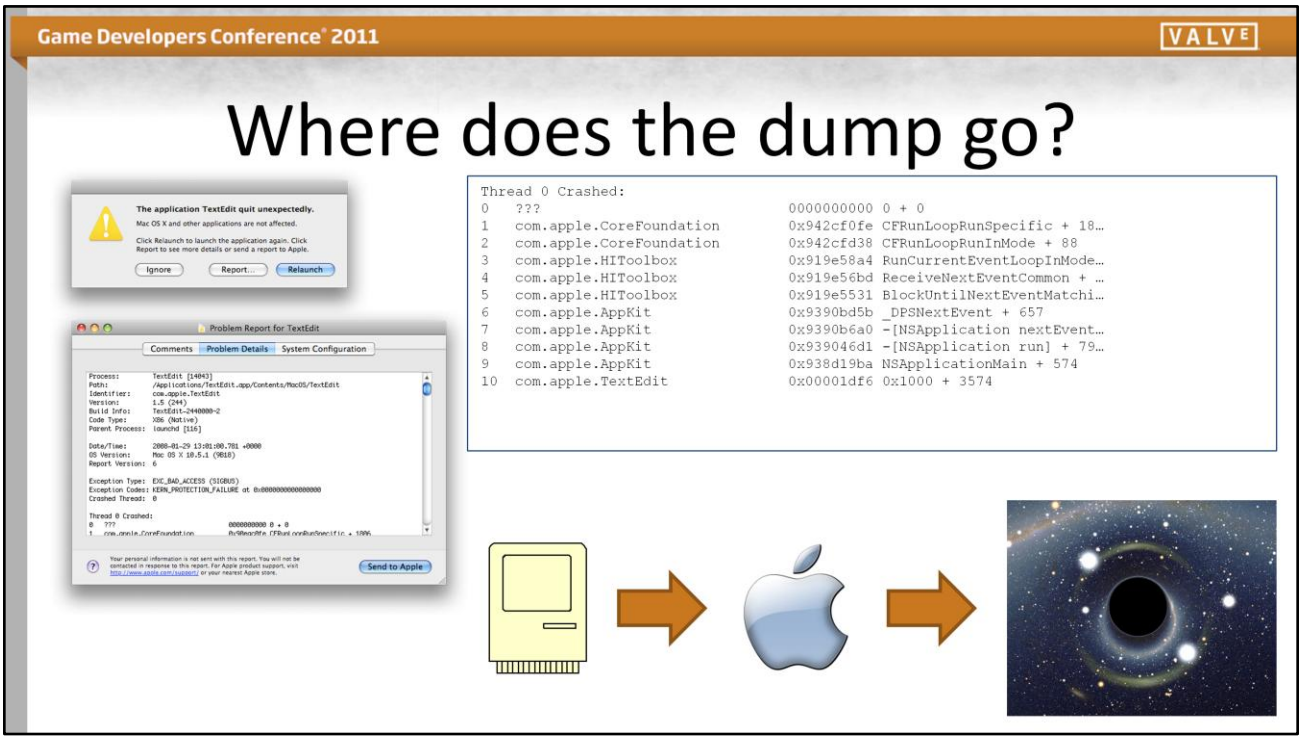


Different things happen after this. Where does the data go?

The simplest possibility is that it just gets written to disk as a big file – either to your local filesystem, or a file server, or wherever. This is convenient for in-house testing, when you can just double click the file and get the debugger to open.



In Windows, it can get sent back to Microsoft via Windows Error Reporting (aka Watson), and you as a developer can have Microsoft send that along to you via Winqual.



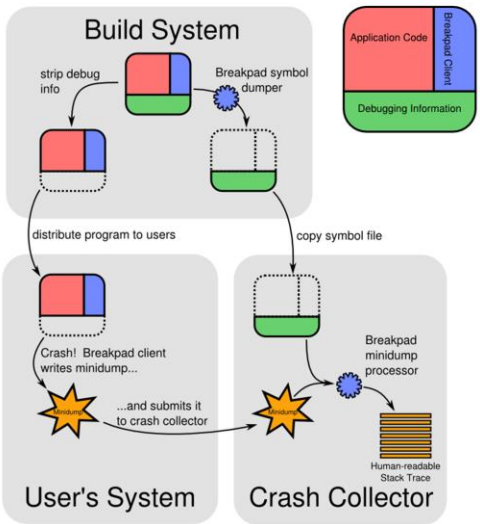
On the Mac platform, app crashes automatically go into MacOS's built in crash reporter app. This collects a call stack (not the entire stack) and writes it to a text file. The customer can push "Report" to upload this text file to Apple, where it falls beneath an event horizon and is never heard from again. They don't give devs access.

Because Mac CRT libraries are often built with symbols included, you'll usually get useful stacks even if you blew up inside the OS.

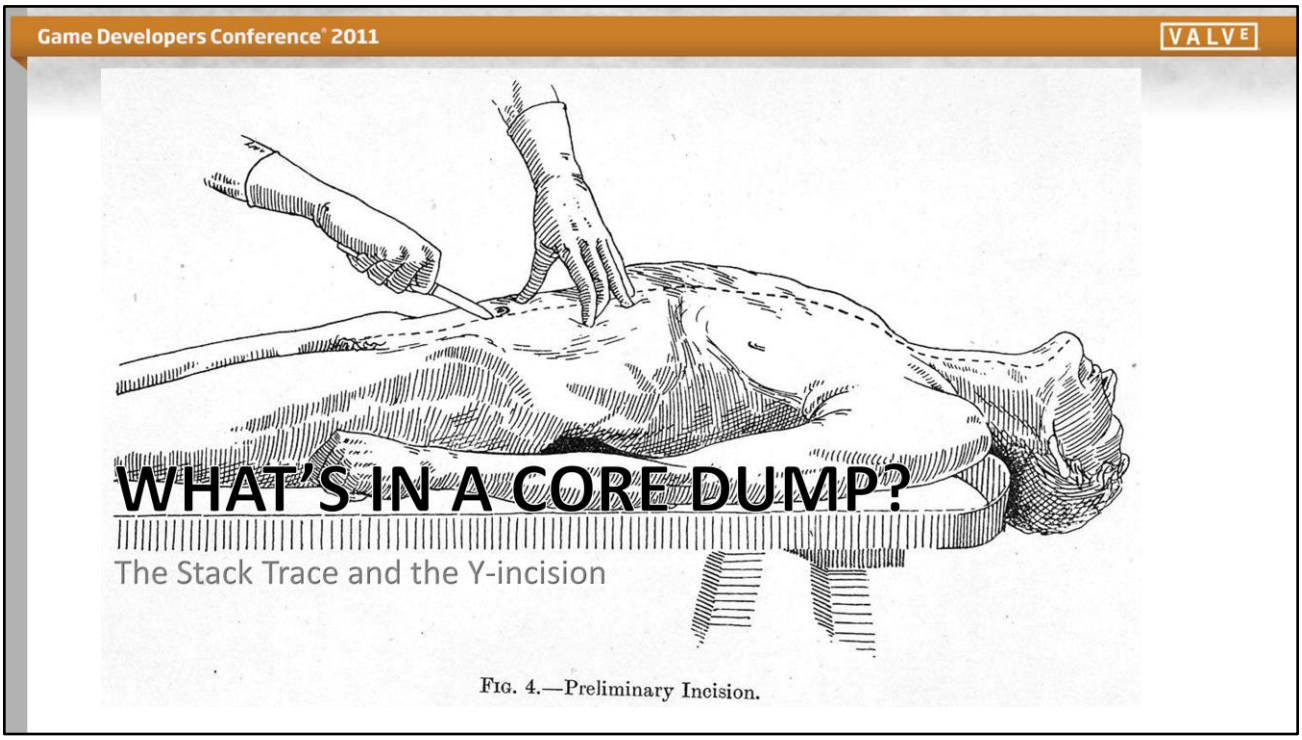
http://developer.apple.com/library/mac/#technotes/tn2004/tn2123.html

Where does the dump go?

- Write your own
- Google Breakpad
- Talkback
- Crashrpt
- etc

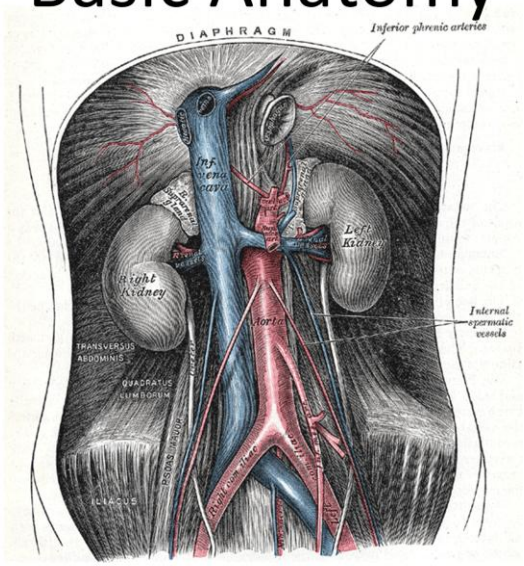


Or you can always write your own handler and do whatever you want -- including sending this data back to you.
Breakpad is one open source solution, which we use, but there are alternatives.
Certain digital distribution services provide this to you

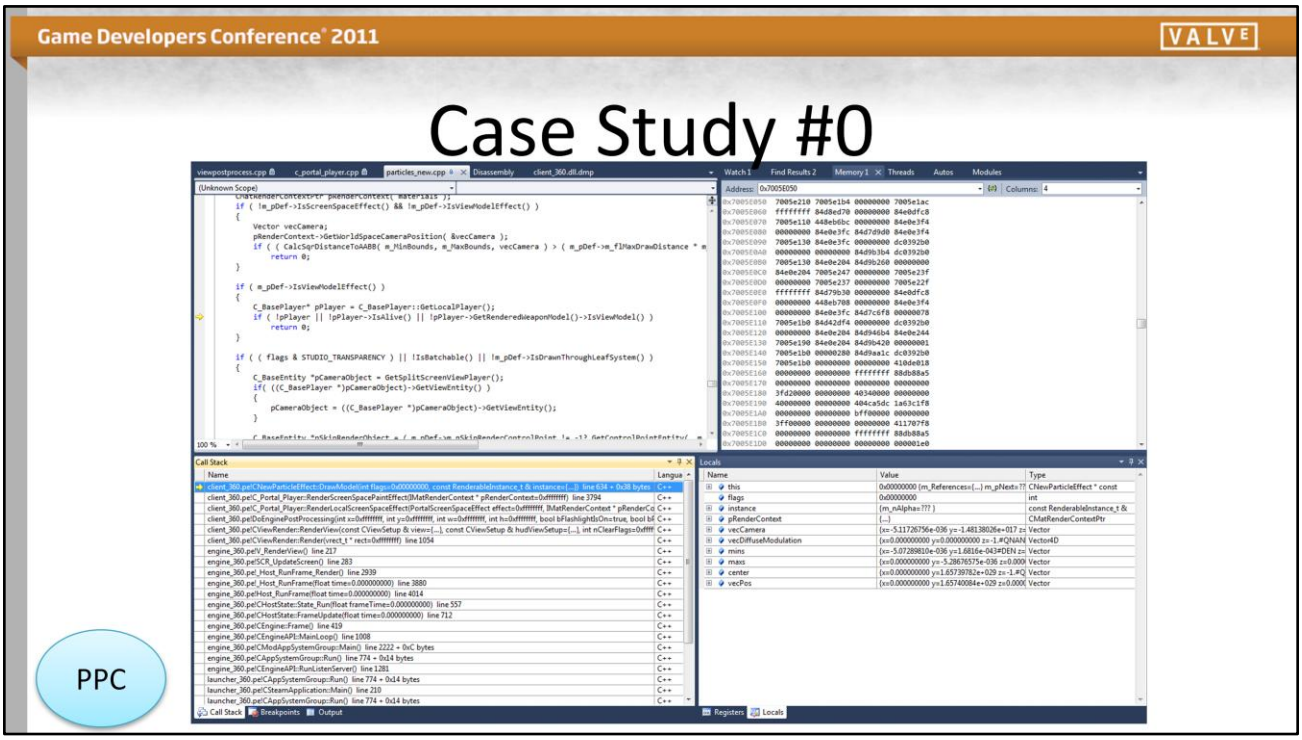


What clues to cause of death can we learn from looking inside the body?

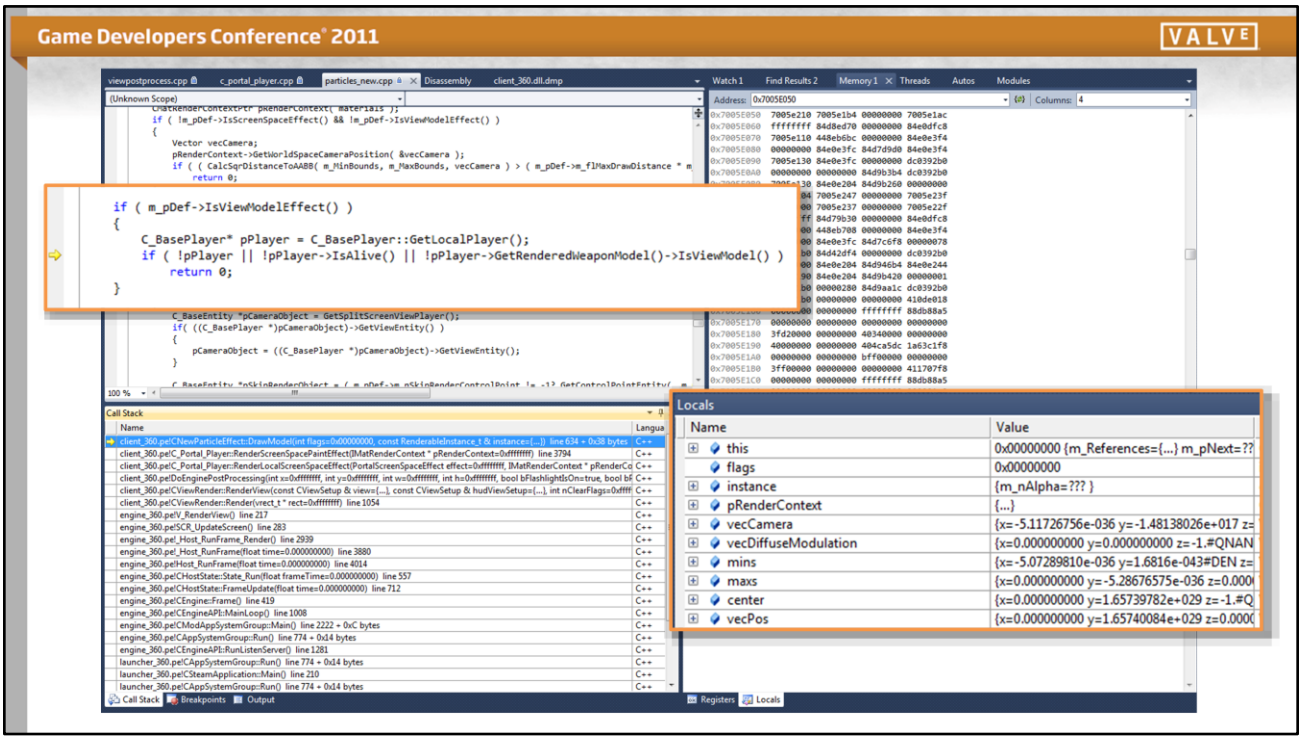
Basic Anatomy



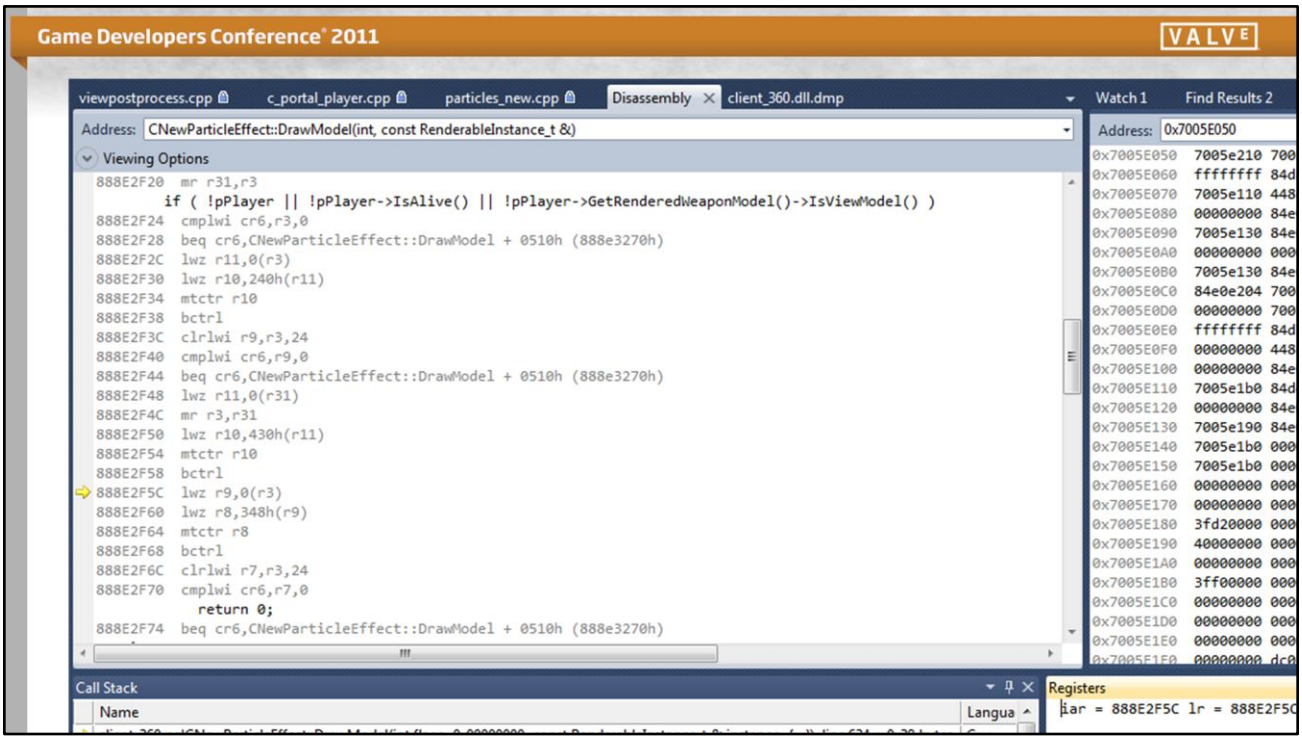
In doing an autopsy, it helps to know where everything is and how it's supposed to be put together generally. In particular, you'll often find yourself manually poking through call stacks to find function parameters. Data has a tendency to ricochet around program memory like a bullet inside the thorax,



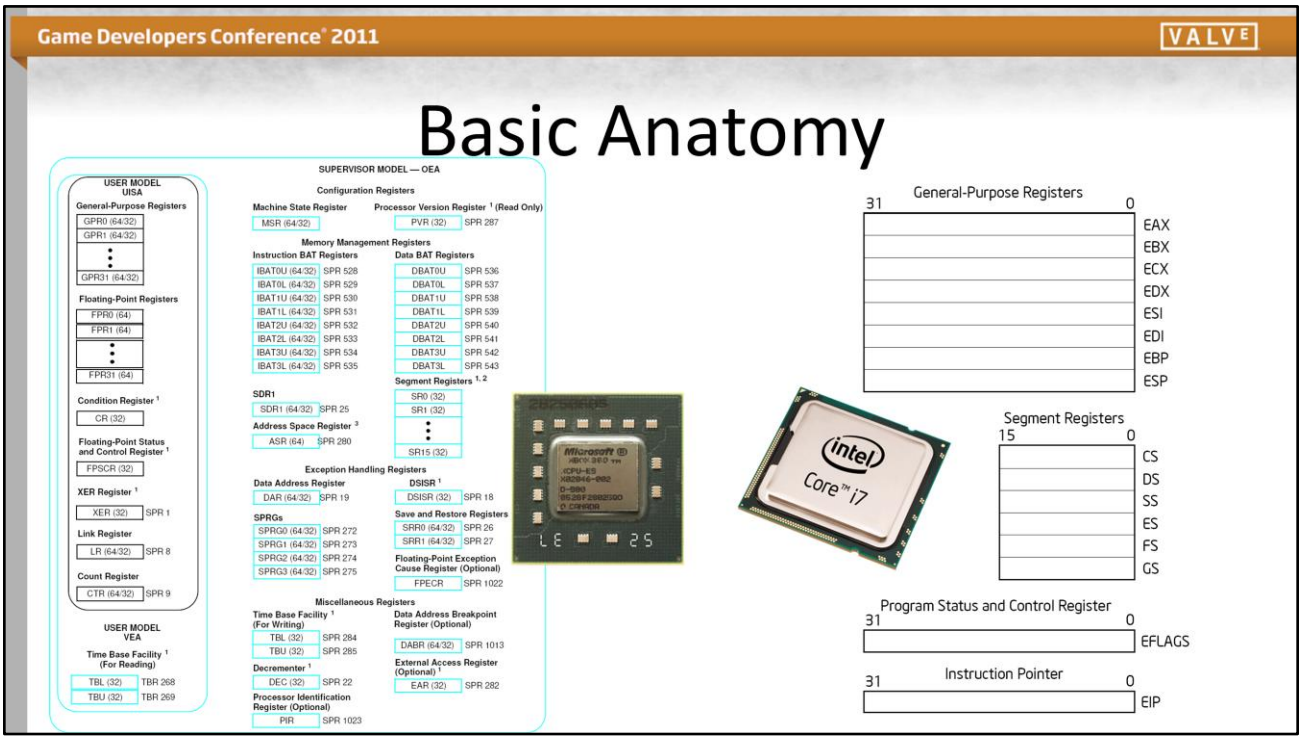
Let’s warm up with a really simple crash that was sitting on my hard drive.



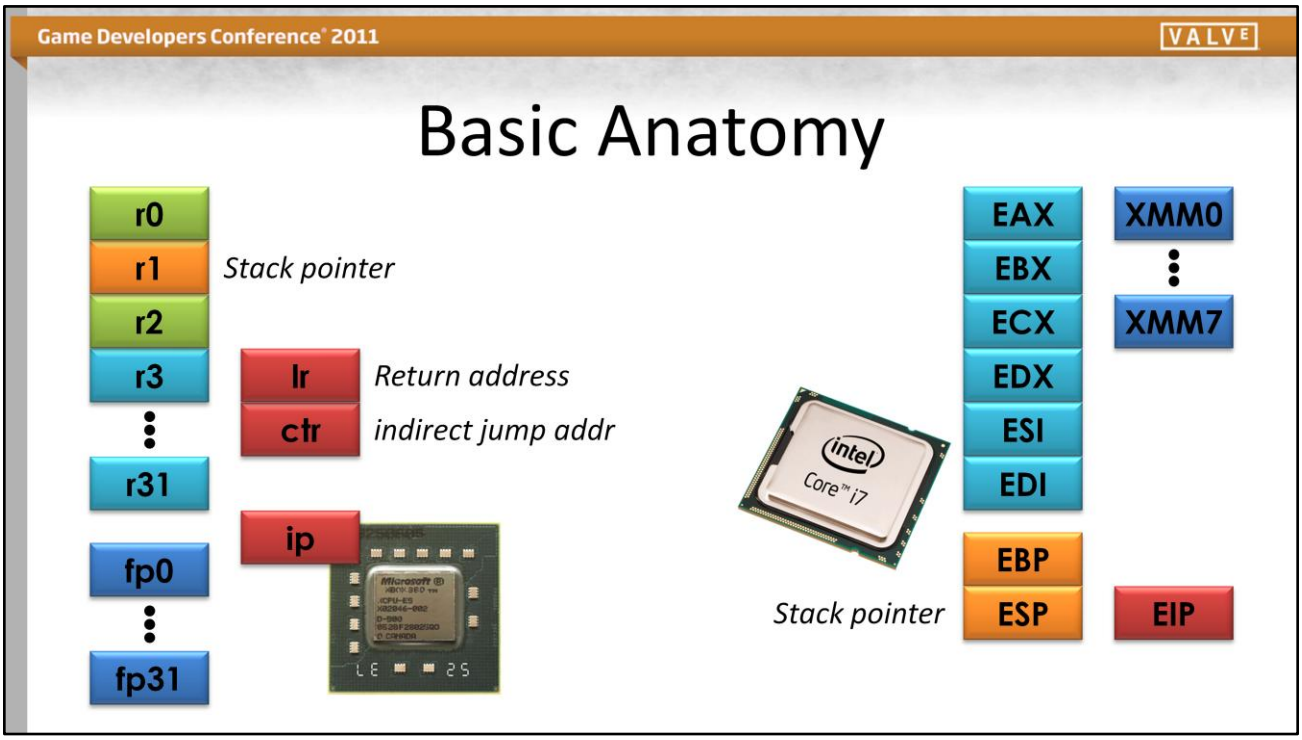
We've crashed somewhere in that block, but where exactly? There's a bunch of places that could go wrong. And, because we've built in release, opening the Locals window doesn't seem to have anything useful for us. So, when the going gets tough, the tough go to the disassembly.



But, as you can see, the disassembly window doesn't always have nice variable names. So to understand the disassembly, you need to understand the CPU's registers. Let's take a quick review.



The two major CPUs game developers deal with are the POWER architecture and the x86. Each contains POWER has a whole bunch of register, of which only those in the left column really matter: the 32 general purpose registers, which store integers and addresses, 32 floating point registers, and the two jump target registers CTR and LR. On the x86, you have eight general purpose registers, of which EBP and ESP are usually reserved for use in managing the stack.



Of the PowerPC's 32 general purpose registers, some have special meanings to the OS, and r1 is almost always reserved for use as the stack pointer. All the stuff I say about the PPC in this talk, by the way, comes from IBM's documentation for their own ABI – the game consoles have slightly different implementations, whose details you can get from your platform documentation, but the operation is similar or analogous.

viewpostprocess.cpp | c_portal_player.cpp | particles_new.cpp | Disassembly | client_360.dll.dmp | Watch 1 | Find Results 2

Address: CNewParticleEffect::DrawModel(int, const RenderableInstance_t &)

Viewing Options

```
888E2F20  mr r31,r3
          if ( !pPlayer || !pPlayer->IsAlive() || !pPlayer->GetRenderedWeaponModel()->IsViewModel() )
888E2F24  cmplwi cr6,r3,0
888E2F28  beq cr6,CNewParticleEffect::DrawModel + 0510h (888e3270h)
888E2F2C  lwz r11,0(r3)
888E2F38  bctrl
888E2F3C  clrlwi r9,r3,24
888E2F40  cmplwi cr6,r9,0
888E2F44  beq cr6,CNewParticleEffect::DrawModel + 0510h (888e3270h)
888E2F48  lwz r11,0(r31)
888E2F54  mtctr r10
888E2F58  bctrl
888E2F5C  lwz r9,0(r3)
888E2F64  mtctr r8
888E2F68  bctrl
888E2F6C  clrlwi r7,r3,24
888E2F70  cmplwi cr6,r7,0
            return 0;
888E2F74  beq cr6,CNewParticleEffect::DrawModel + 0510h (888e3270h)
```

r3 = 00000000

Address: 0x7005E050

```
0x7005E050  7005e210 700
0x7005E060  ffffffff 84d
0x7005E070  7005e110 448
0x7005E080  00000000 84e
0x7005E090  7005e130 84e
0x7005E0A0  00000000 000
0x7005E0B0  7005e130 84e
0x7005E0C0  84e0e204 700
0x7005E0D0  00000000 700
0x7005E0E0  ffffffff 84d
0x7005E0F0  00000000 448
0x7005E100  00000000 84e
0x7005E110  7005e1b0 84d
0x7005E120  00000000 84e
0x7005E130  7005e190 84e
0x7005E140  7005e1b0 000
0x7005E150  7005e1b0 000
0x7005E160  00000000 000
0x7005E170  00000000 000
0x7005E180  3fd20000 000
0x7005E190  40000000 000
0x7005E1A0  00000000 000
0x7005E1B0  3ff00000 000
0x7005E1C0  00000000 000
0x7005E1D0  00000000 000
0x7005E1E0  00000000 000
```

Call Stack | Name | Langua

Registers

iar = 888E2F5C lr = 888E2F5C

We can return now to the disassembly window and try to get a better idea of where the game died. Notice that there are two OR operators in that conditional expression. That means the corresponding assembly ought to have two corresponding branch opcodes (because of C++'s early-out shortcut semantics). You can see that the instruction which actually crashed came after both branches, so the crash occurred somewhere in pPlayer->GetRenderedWeaponModel()->IsViewModel(). In particular, the op that died was trying to load a word from the address in register 3, but r3 contained NULL. Why? Let's see what it means for one function to pass data to another in C++.

int foo(int a, int f, int *pb)

```
int foo( int a, int f, int *pb )
{
   return a + f + *pb;
}


void caller()
{
   int c = 2;
   printf( "%d\n",
     foo( globalint, 1, &c ); );
}
```

```
foo(int, int, int*):
stdu      r1,-0x40(r1)   ; move stack for locals
 ; function does its work..
extsw     r3,r3          ; return value is on r3
addi      r1,r1,0x40     ; restore stack pointer
blr


caller:
mr        r3,r30         ; int a is first parameter, r3
li        r4,1           ; load "1" into int b, which is r4
addic     r5,r1,0x70     ; load r5 with a pointer to a stack
variable
bl        foo(int, float, int*)
 ; return value comes back on r3
```

PPC

On the PowerPC, parameters are passed from left to right, on registers r3 through r10. Params too big to go on registers go on the stack.

Float CThing::foo(int a, float f, int &pb)

```
struct CThing
{
   int m_n;
   CThing( int a ) : m_n( a ) {};
   float foo( float f, int &pb )
   {
     return m_n + pb + f;
   }
};

void Caller( int a, int c )
{
   int b = c;
   CThing woo(a);
   float f = woo.foo( 3.14f, b );
   Msg("%f\n", f);
}
```

```
CThing::foo:
stdu     r1,-0x40(r1)
lwz      r3,0x0(r3)   ; load this->m_n
lwz      r4,0x0(r5)   ; load through pb ref
addc     r3,r3,r4     ; m_n + pb
std      r3,0x30(r1)  ; move data from an int register...
lfd      f2,0x30(r1)  ; ...to a float register via stack
fcfid    f2,f2        ; converts int to float
fadds    f1,f2,f1     ; ( ... + f )
addi     r1,r1,0x40
blr                   ; return

Caller:
lis      r4,0x0
stw      r3,0x70(r1)
addic    r5,r1,0x70
addi     r4,r4,0x0
stw      r30,0x74(r1)
addic    r3,r1,0x74
lfs      f1,0x0(r4)
bl       CThing::foo(float, int&)
; return value comes back on f1
```

PPC

C++ member functions have an invisible first parameter “this”, which is a pointer to the class instance. Thus, in a C++ function, r3 will always have THIS, and the first formal parameter goes on r4.

int foo(int a, int f, int *pb)

```
int foo( int a, int f, int *pb )
{
   return a + f + *pb;
}


void caller()
{
   int c = 2;
   printf( "%d\n",
     foo( globalint, 1, &c ); );
}
```

```
foo(int, int, int*):
push        ebp
mov         ebp, esp
mov         eax, DWORD PTR _pb$[ebp]
mov         edx, DWORD PTR [eax]
add         edx, DWORD PTR [ecx]
mov         DWORD PTR tv134[ebp], edx
fild        DWORD PTR tv134[ebp]
fadd        DWORD PTR _f$[ebp]
pop         ebp
ret         8


caller:
;; push params right to left
lea         eax, DWORD PTR _c$[ebp] ;; get address to c
push        eax     ; push &c onto stack
push        1       ; push immediate 1 onto stack
push        edi     ; push globalint onto stack
call        ?foo@@YAHHHPAH@Z                          ; foo
;; return value comes back on eax
```

x86

On the x86, there's many different calling conventions, but in most, parameters are pushed onto the stack in right to left order.

Float CThing::foo(int a, float f, int &pb)

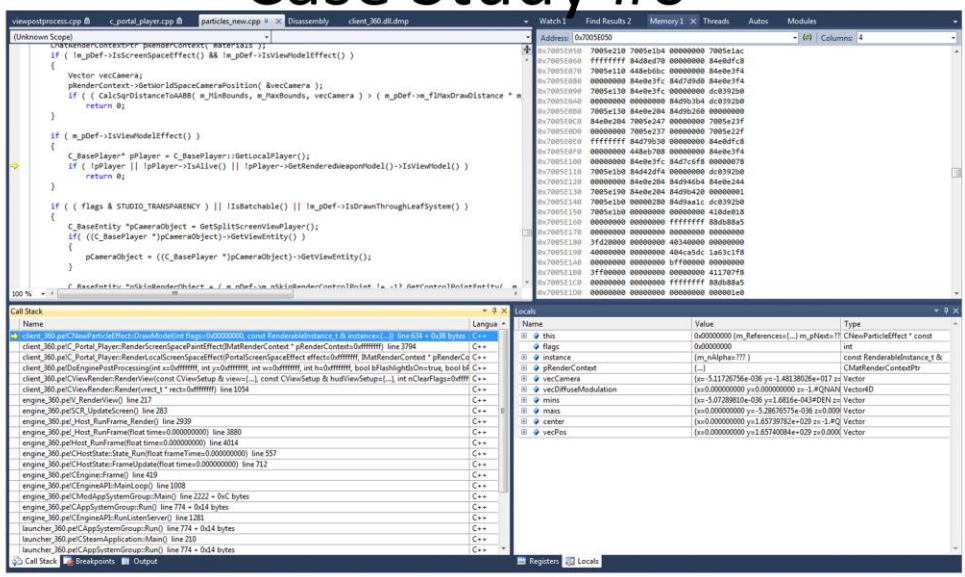
```
struct CThing
{
   int m_n;
   CThing( int a ) : m_n( a ) {};
   float foo( float f, int &pb )
   {
     return m_n + pb + f;
   }
};

void Caller( int a, int c )
{
   int b = c;
   CThing woo(a);
   float f = woo.foo( 3.14f, b );
   Msg("%f\n", f);
}
```

x86

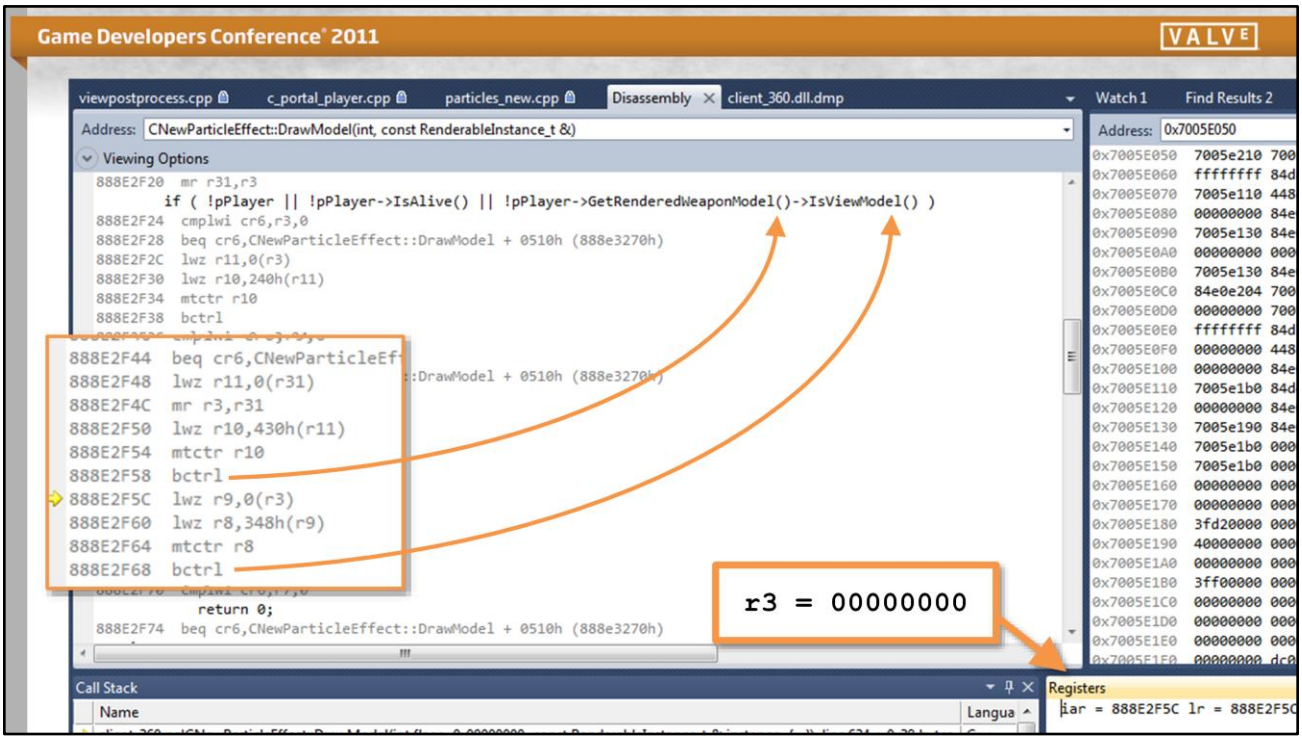
```
CThing::foo:
push        ebp
mov         ebp, esp
mov         eax, DWORD PTR _pb$[ebp]
mov         edx, DWORD PTR [eax]
add         edx, DWORD PTR [ecx]
mov         DWORD PTR tv134[ebp], edx
fild        DWORD PTR tv134[ebp]
fadd        DWORD PTR _f$[ebp]
pop         ebp
ret         8

Caller:
fld         DWORD PTR __real@4048f5c3 ; 3.14 const
mov         DWORD PTR _b$[ebp], eax
lea         eax, DWORD PTR _b$[ebp]
push        eax
push        ecx
lea         ecx, DWORD PTR _woo$[ebp] ;; this pointer
fstp        DWORD PTR [esp]
mov         DWORD PTR _woo$[ebp], edi
call        ?foo@CThing@@QAEMMAAH@Z
 ; return value comes back on x87 fpu stack
```

There's a few common x86 conventions for dealing with C++ member functions. The more efficient one, THISCALL, passes "this" on the ecx register. Others simply pass it as an implicit first parameter on the stack.

Case Study #0



So, armed with this knowledge, we can return to our case study.



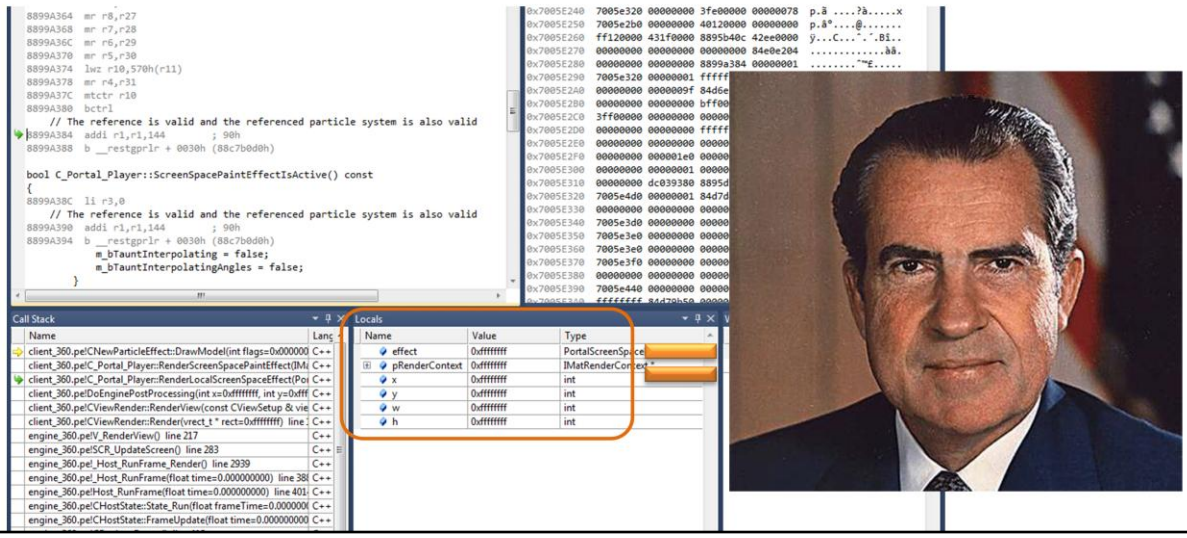
In this case, we've gone past the two conditionals, and died between the bctrls. We know that the first bctrl returned 0x00, and that its return value was to become THIS to the second, so we can look up at the two virtual function calls in the third clause of the if(), and see that what must have happened is that GetRenderedWeaponModel() returned NULL.

WHY THE DEBUGGER LIES TO YOU

And why forensics is better than relying on eight-bit stoolpigeons

Why the debugger is a filthy lying rat.

The Watch Window: Unreliable witness



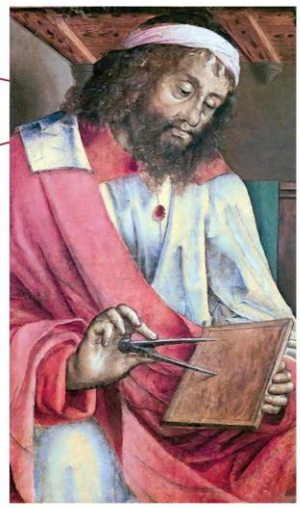
As you've noticed, the watch window can be what's technically referred to as a BIG FAT LIAR. To see the reason for this, we can look at what the optimizing compiler does to a function when you compile it in release mode.

Pictured: the "0xfffffffffff" lie for an unknown local

Where are my locals?

r3 r4

```
// integer common divisor
int euclid( int x, int y )
{
   int a = x;
   int b = y;

   if ( a == 0 )
      return b;

   while ( b != 0 )
   {
      if ( a > b )
         a = a - b;
      else
         b = b - a;
   }
   return a;
}
```



```
; while ( b != 0 )
lwz          r3,b$(r1);
cmpwi        cr6,r3,0;
bne          cr6,$LN3@euclid
; }
addi         r1,r1,80 ; reset stack pointer
mtlr         r12,$(r1)
mtcr         r12,68(r1)
lmw          r26,-40(r1)
blr ;; return r3 ; $LN2@euclid move stack pointer
; a = a - b;
lwz          r3,a$(r1);
lwz          r4,b$(r1);
subf         r11,r3,r4
stw          r11,a$(r1);
lwz          r8,a$(r1);
cmpwi        cr6,r8,0
bne          $LN2@euclid
$LN2@euclid: cr6,$LN4@euclid
; b = b - a;
lwz          r10,b$(r1);
lwz          r9,a$(r1);
subf         r8,r9,r10;
stw          r8,b$(r1);
b            $LN3@euclid ;; the end of the function
$LN4@euclid:
; }
b            $LN4@euclid
$LN5@euclid:
```

PPC

Euclid's Greatest Common Divisor algorithm

Here's the simplest function I could think of, Euclid's Greatest Common Divisor algorithm. Based on the code you would expect to find room for two locals on the stack, and indeed that's what we've got. Looking at the code the "debug" compiler produced, we see pretty much the prolog code we expect: it makes room on the stack and moves the input parameters from r3 and r4 to the locals A and B in memory. When it does the math, you see that it loads A and B from memory into a register before each operation, does the op, and then stores them back. Finally, it retrieves the result from memory, puts it onto r3 for return, and restores the stack.

The compiler does this in "debug" builds in order to preserve a 1:1 mapping between the source code and the machine language. The assembly performs work in the same order as the source, and for every line of source code you can find the corresponding machine op, and vice versa. Also, the debugger can always find local variables in memory, because they are always stored back after being modified, and always to consistent locations.

You might expect that all this round trip traffic to main RAM might be less than performant, and you'd be right. Let's look at that same function compiled in Release.

In release:

r3 r4

```
// integer common divisor
int euclid( int x, int y )
{
   int a = x;
   int b = y;

   if ( a == 0 )
      return b;

   while ( b != 0 )
   {
      if ( a > b )
         a = a - b;
      else
         b = b - a;
   }
   return a;
}
```

0x14	caller frame
0x10	backchain...
0x0c	int a
0x08	int b
0x04	LR save
+ 0x00	+0x10

```
; Begin code for function
; {
mr          r10,r3  ; int a = x;
mr          r11,r4  ; int b = y;
; if ( a == 0 )
cmpwi       cr6,r3,0
bne         cr6,$LN8@euclid
mr          r3,r4
blr;  return b
$LN8@euclid:  ;; else
; while ( b != 0 )
cmpwi       cr6,r11,0
beq         cr6,$LN3@euclid
$LL4@euclid: ; Start of loop
; {
; if ( a > b )
cmpw        cr6,r10,r11
ble         cr6,$LN2@euclid
subf        r10,r11,r10 ; a = a - b;
; else
b           $LN1@euclid
$LN2@euclid:
subf        r11,r10,r11 ; b = b - a;
$LN1@euclid:
cmpwi       cr6,r11,0
bne         cr6,$LL4@euclid
$LN3@euclid:
mr          r3,r10
; }
blr ; return a
; End code for function
```

PPC

As you can see, now there is no stack! The function doesn't even have to move the stack pointer since it calls no sub functions, and more importantly, it never stores its locals in memory. They are always in the registers. So, the debugger's watch may not know where to look for these numbers. Also, generally, the compiler can do many kinds of optimization that may cause some intermediate values to vanish altogether. It can order machine code differently from the source, collapse common calculation, all sorts of things that destroy the 1:1 mapping between source and assembly. That's why the instruction arrow jumps around a lot when stepping through release code, and why the debugger has such a hard time finding local data: the data is never in memory in the first place, but stored in registers, or possibly never even stored at all.

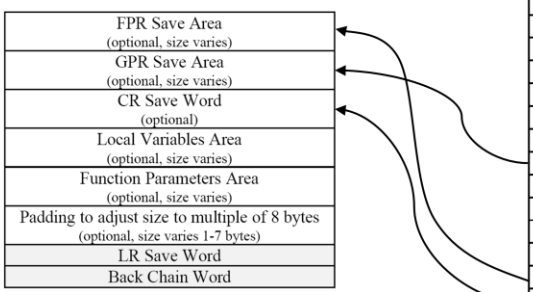
Who can you trust?

Unreliable witness?

Try SCIENCE!

The watch window has to interpret the compiler results it sees to give you data, and when its interpretations fail, it's an unreliable witness. As any CSI knows, if you've got an unreliable witness, the best place to turn is the physical evidence.

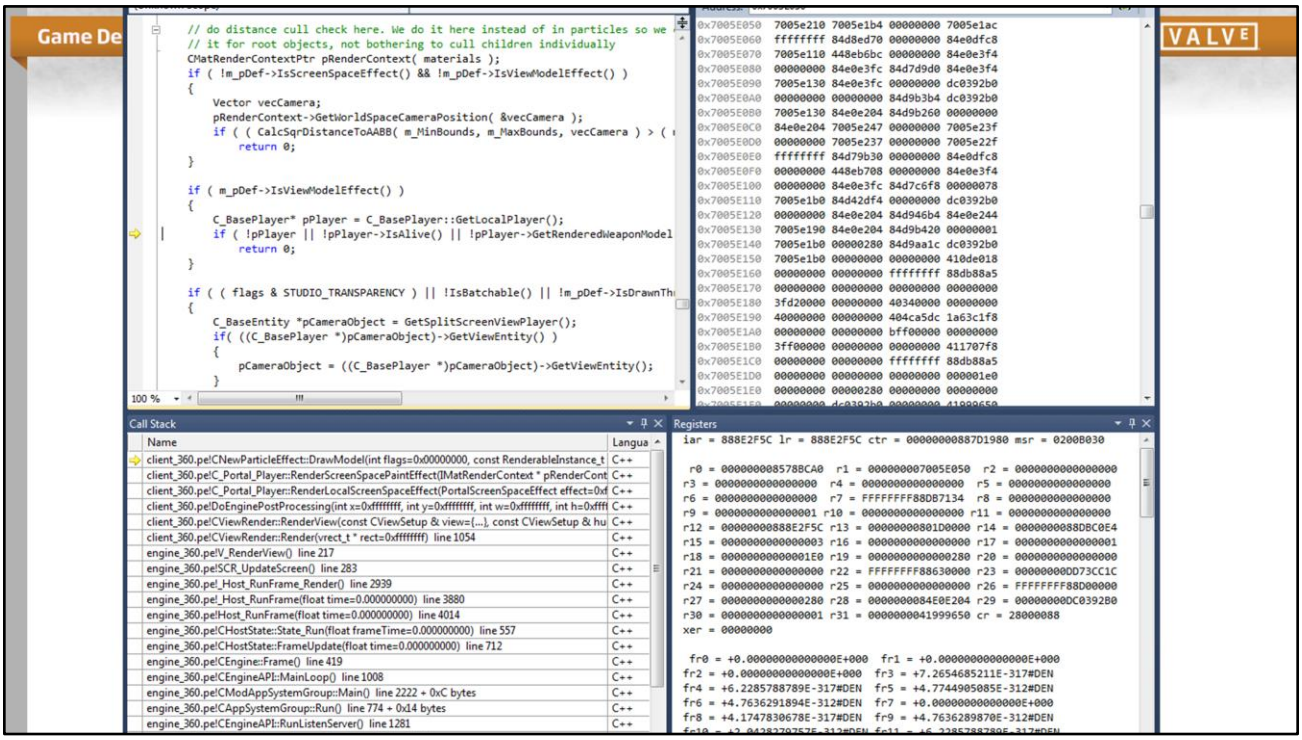
PowerPC nonvolatile registers



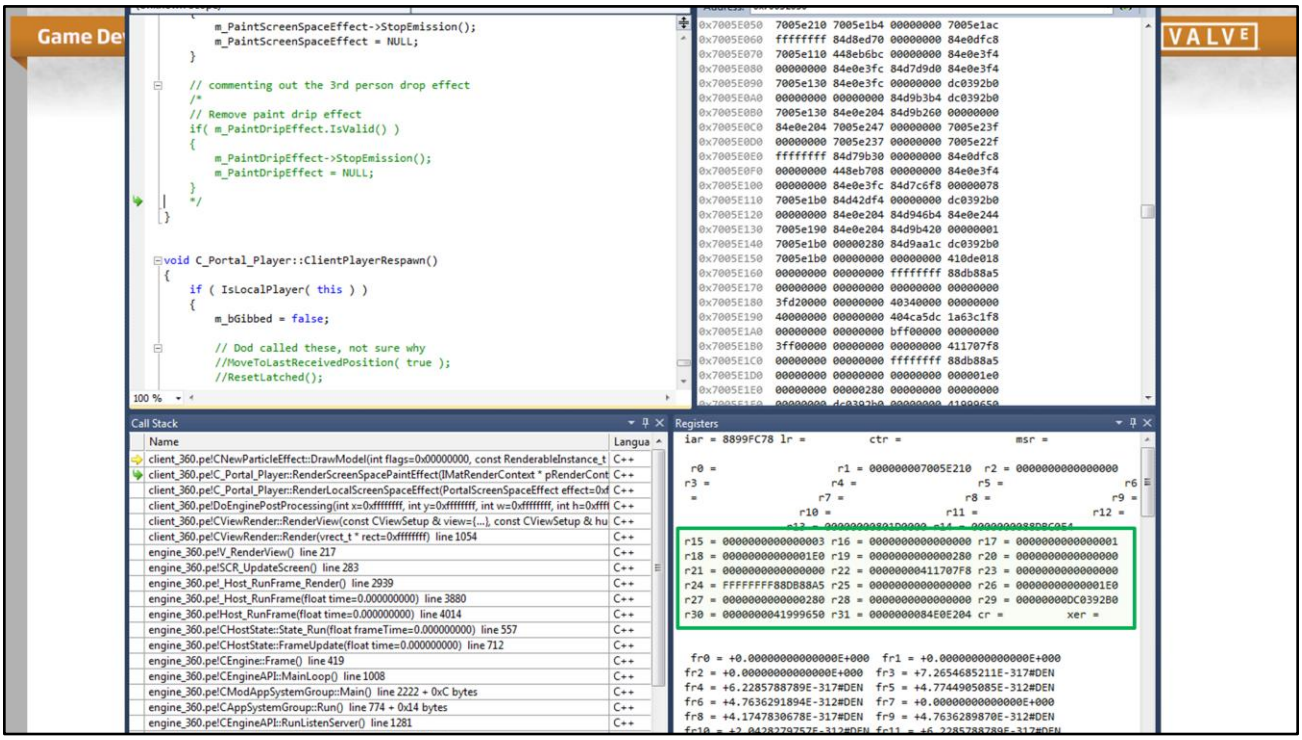
PowerPC EABI register usage

Register	Type	Used for:
R0	Volatile	Language Specific
R1	Dedicated	Stack Pointer (SP)
R2	Dedicated	Read-only small data area anchor
R3 - R4	Volatile	Parameter passing / return values
R5 - R10	Volatile	Parameter passing
R11 - R12	Volatile	
R13	Dedicated	Read-write small data area anchor
R14 - R31	Nonvolatile	
F0	Volatile	Language specific
F1	Volatile	Parameter passing / return values
F2 - F8	Volatile	Parameter passing
F9 - F13	Volatile	
F14 - F31	Nonvolatile	
Fields CR2 - CR4	Nonvolatile	
Other CR fields	Volatile	
Other registers	Volatile	

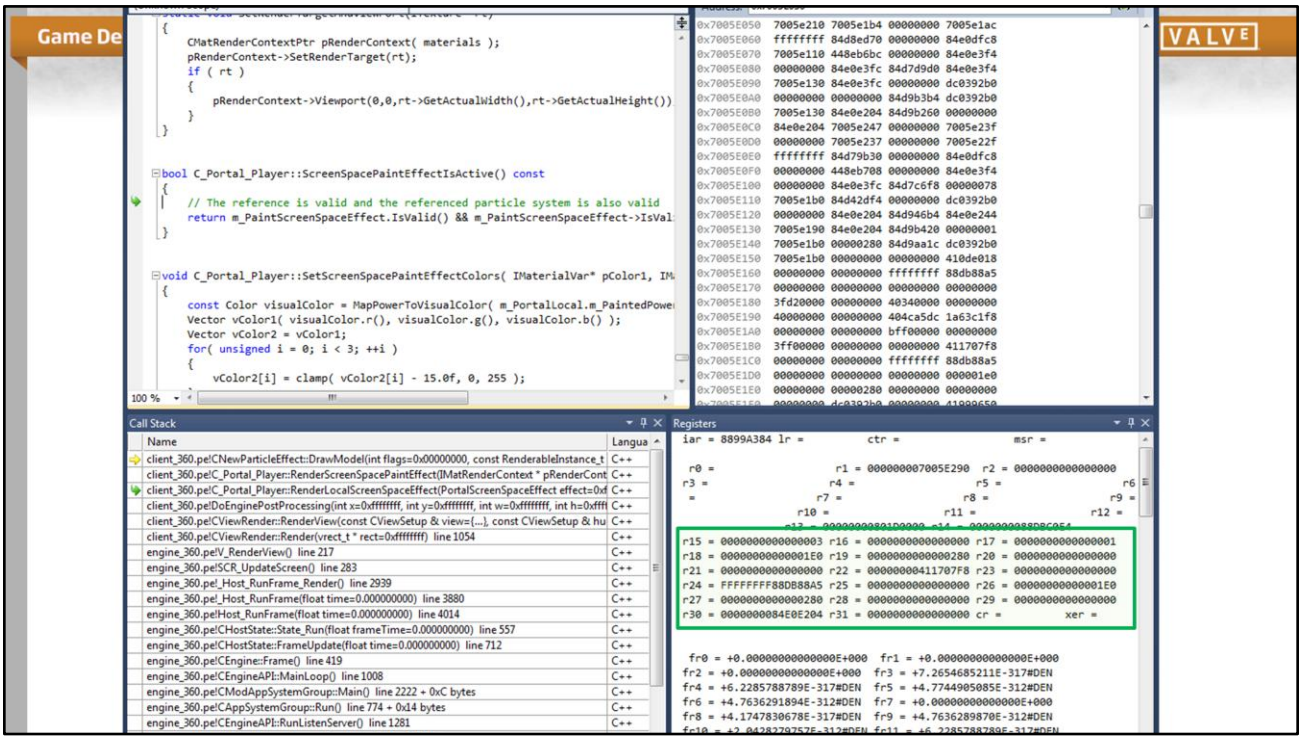
The notion of volatile and nonvolatile registers on the PPC is very useful in debugging analysis. Because a function must restore nonvolatile regs to their initial state before returning to its caller, that means that it must always save them to the stack before modifying them. That means that nonvolatile regs are almost always recoverable so long as you still have the stack. Volatile registers can easily be overwritten.



The debugger can show you all the registers in their current state at the bottom frame of the stack, because they were stored in the exception at the moment of crashing..

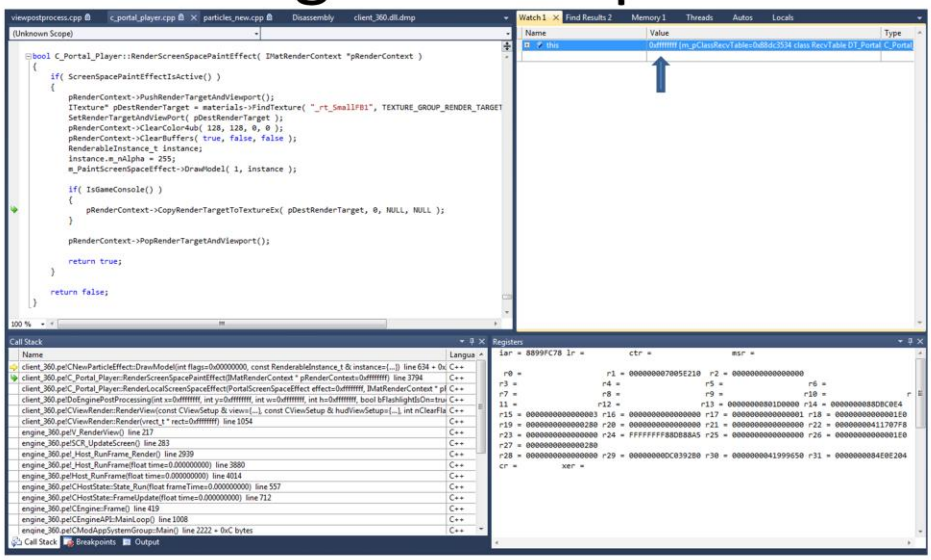


… and is usually pretty good at recovering the nonvolatile registers from the other frames of the stack, again because they must have either been untouched, or stored in memory somewhere.

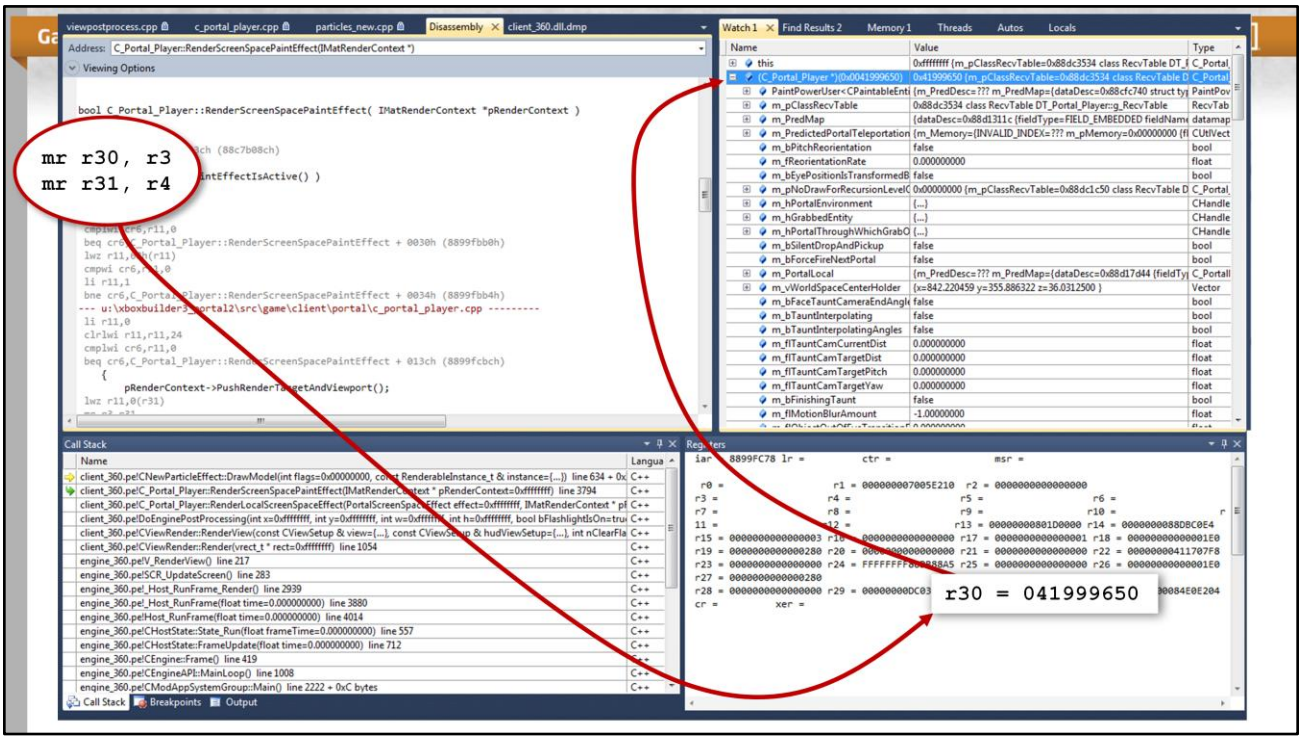


That's what it means to set the "active frame" in the call stack window. It's not just about moving the contents of the source pane to the right place – it tells the debugger to update the stack, local, and register windows to what they would have been in that context. As I move up and down in the call stack pane, it automatically updates r1 in the registers pane to contain the stack base for that function.

Finding a `this` pointer



So let's say we wanted to recover THIS from one frame up. The watch window lies and says THIS is 0xffffffff.



And that's because the r3 register is volatile and gone by this point. However, if you look carefully at the disasm, you'll see that one of the first things this function did was move the contents of r3 (which you'll recall is where the "this" parameter gets passed) onto r30. R30 is nonvolatile, so if we can find it in the registers pane, it should be possible to punch it into the watch window typecast as the appropriate pointer type, and bang it's like your locals window actually worked properly.

Function Parameters are often on Nonvolatile Registers

viewpostprocess.cpp | c_portal_player.cpp | particles_new.cpp | Disassembly | client_360.dll.dmp

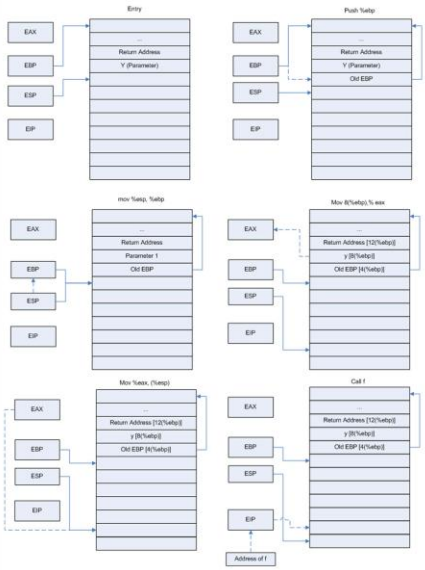
Address: C_Portal_Player::RenderLocalScreenSpaceEffect(PortalScreenSpaceEffect, IMatRenderContext *, int, int, int, int)

Viewing Options

```
bool C_Portal_Player::RenderLocalScreenSpaceEffect( PortalScreenSpaceEffect effect, IMatRenderContext *pRenderContext, int x, int
{
 mflr r12
 bl __savegprlr + 0030h (88c7b080h)
 stwu r1,-90h(r1)
$M249798:
 mr r31,r3
    C_Portal_Player *pLocalPlayer = C_Portal_Player::GetLocalPlayer();
 li r3,-1          ; 0FFFFh
 mr r30,r4
 mr r29,r5
 mr r28,r6
 mr r27,r7
 mr r26,r8
 bl C_BasePlayer::GetLocalPlayer (887d1058h)
    if( pLocalPlayer )
 cmplwi cr6,r3,0
 beq cr6,C_Portal_Player::RenderLocalScreenSpaceEffect + 0064h (8899a38ch)
    {
        return pLocalPlayer->RenderScreenSpaceEffect( effect, pRenderContext, x, y, w, h );
 lwz r11,0(r3)
 mr r9,r26
 mr r8,r27
 mr r7,r28
 mr r6,r29
 mr r5,r30
 lwz r10,570h(r11)
 mr r4,r31
 mtctr r10
 bctrl
```

Generally speaking most compilers like to move function params into nonvolatiles. They don't always do it, they don't always do it at the same time or in the same place or in the same order or onto the same registers, so you'll need to look at the disasm to see if it happened in the current frame and if so where the parameters went. However, often you'll get lucky and find your parameters or locals on a nonvolatile somewhere, which means that you can frequently retrieve missing function params in this way.

x86 calling convention



address	use
0x10000	Function A parameter 1
0xfffc	return address before function A
0xfff8	old EBP before function A
0xfff4	Function B parameter 1
0xfff0	Function B parameter 2
0xffec	return address to function A
0xffe8	function A's EBP
0xffe4	local variable
0xffe0	local variable
0xffdc	local variable
0xffd8	Function C parameter 1
0xffd4	return address to function B
0xffd0	function B's EBP

- There are many
- Caller restores ESP
  - cdecl
- Callee restores ESP
  - stdcall
  - fastcall
  - thiscall (C++ `this` pointer goes on ECX)

The x86 has many different calling conventions, which mostly differ in whether the caller or the callee restores the stack pointer, and what order parameters go onto the stack in. They're numerous and well documented elsewhere, so I'll skip past them.

Diagram credit: Jerry Coffin

http://en.wikibooks.org/wiki/X86_Disassembly/The_Stack

Game Developers Conference® 2011 VALVE

In x86, everything is in memory

```
; static bool __cdecl C_Portal_Player::RenderLocalScreenSpaceEffect(enum,class IMatRenderContext *,int,int,int,int)

  _TEXT SEGMENT
  _effect$ = 8
  _pRenderContext$ = 12
  _x$ = 16
  _y$ = 20
  _w$ = 24
  _h$ = 28
  ; : {
  push ebp
  mov ebp, esp
  push -1
  call ?GetLocalPlayer@C_BasePlayer@@SAPAV1@H@Z
  add esp, 4
  test eax, eax
  je SHORT $LN1@RenderLoca

  mov ecx, DWORD PTR _h$[ebp]
  mov edx, DWORD PTR [eax]
  mov edx, DWORD PTR [edx+1392]
  push ecx
  mov ecx, DWORD PTR _w$[ebp]
  push ecx
  mov ecx, DWORD PTR _y$[ebp]
  push ecx
  mov ecx, DWORD PTR _x$[ebp]
  push ecx
  mov ecx, DWORD PTR _pRenderContext$[ebp]
  push ecx
  mov ecx, DWORD PTR _effect$[ebp]
  push ecx
  mov ecx, eax
  call edx
  pop ebp
  ret 0
  $LN1@RenderLoca:
  or al, al
  pop ebp
  ret 0
```

ebp+24

On the x86, all parameters always get pushed onto the stack anyway, so they're always in memory. Thus the debugger usually does a pretty good job of retrieving then. Even if it doesn't, you can always go poking around in memory yourself to find them, although your eyes may bleed a bid after combing through the disassembly to try to figure out which offset corresponds with which value.

Debugging live "release builds" is just like debugging a crash

- Watch window lies same way in both cases
- Really the same skills
- Diagnose bugs in situ without needing repro
- Live processes have a little more info
- Connect to QA kits remotely from your desk

Everything I've described just now isn't specific to crashes. All of this is really as much true for debugging any optimized executable, regardless of whether it's dead or alive. So, you can use these same skills to attach to a currently running game – even a release image running on a QA kit – and try to figure out a problem without having to reproduce it on your machine in a debug exe.

"Get this back to my lab"

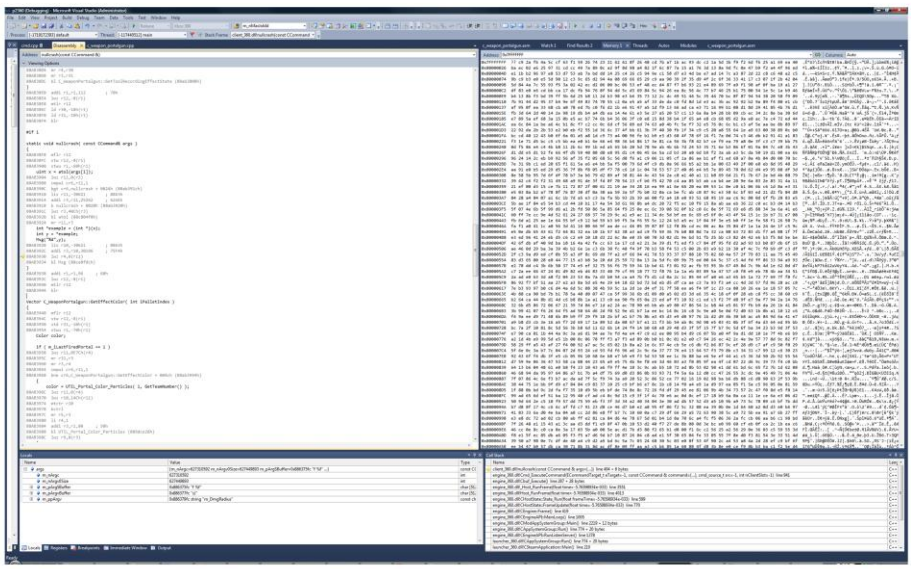
Also, if you have a live issue on a running kit, but for some reason it's inconvenient for you to debug it at just that moment – maybe you're busy, or QA needs the kit back, or it's going to take a long time – then you can manually trigger a dump to create a state snapshot that you can take back to your workstation and examine at your leisure. There's mechanisms for this on every platform to do this from the debugger, and programmatically from inside the game.

COMMON PATTERNS

Every Bug Has Its Fingerprint

Let's look at some common issues and how to recognize them.

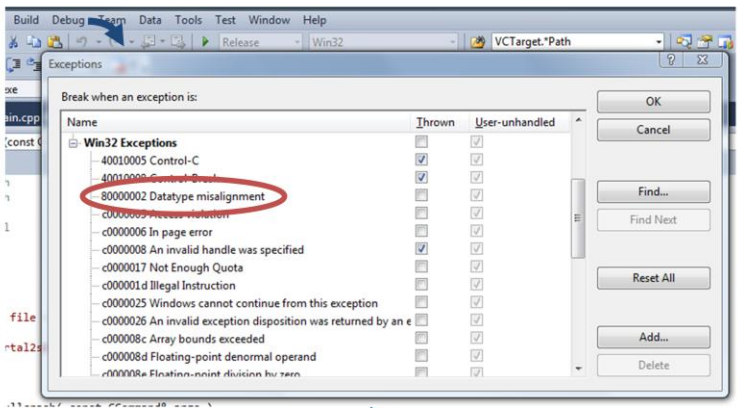
Misaligned read



A very simple one: misaligned read – trying to, say, load a 32-bit word from an address that isn't a multiple of four bytes. (That's a crash on PPC.)

Misaligned read

```
static void nullcrash( const CCommand& args )
{
88AB38F8  mflr r12
88AB38FC  stw r12,-8(r1)
88AB3900  stwu r1,-60h(r1)
    uint x = atol(args[1]);
88AB3904  lwz r11,0(r3)
88AB3908  cmpwi cr6,r11,1
88AB390C  bgt cr6,nullcrash + 0024h (88ab391ch)
88AB3910  lis r11,-30621        ; 8863h
88AB3914  addi r3,r11,25262     ; 62AEh
88AB3918  b nullcrash + 0028h (88ab3920h)
88AB391C  lwz r3,40Ch(r3)
88AB3920  bl atoi (88c804f0h)
88AB3924  mr r11,r3
    int *example = (int *)(x);
    int y = *example;
    Msg("%d",y);
88AB3928  lis r10,-30621        ; 8863h
88AB392C  addi r3,r10,30196     ; 75F4h
88AB3930  lwz r4,0(r11)
88AB3934  [illegible]Msg (88ce8fdch)
}
88AB3938  addi [illegible]r1,96     ; 60h
88AB393C  lwz r12, [illegible]1)
88AB3940  mtlr r12
88AB3944  blr
```



"Unhandled exception at 0x88AB3930 in E:\portal2\default.xex: 0x80000002: Datatype misalignment."

PPC

Okay, that's pretty easy. MSVC will tell you, misaligned read. Before we move on, take a quick look at what those two numbers mean: one is the instruction of the faulty address, and the other is the exception code. You can map numeric exception codes to strings by looking at the "exceptions" dialog under the debug menu.

Misaligned read

```
static void nullcrash( const CCommand& args )
{
88AB38F8  mflr r12
88AB38FC  stw r12,-8(r1)
88AB3900  stwu r1,-60h(r1)
    uint x = atol(args[1]);
88AB3904  lwz r11,0(r3)
88AB3908  cmpwi cr6,r11,1
88AB390C  bgt cr6,nullcrash + 0024h (88ab391ch)
88AB3910  lis r11,-30621        ; 8863h
88AB3914  addi r3,r11,25262     ; 62AEh
88AB3918  b nullcrash + 0028h (88ab3920h)
88AB391C  lwz r3,40Ch(r3)
88AB3920  bl atoi (88c804f0h)
88AB3924  mr r11,r3
    int *example = (int *)(x);
    int y = *example;
    Msg("%d",y);
88AB3928  lis r10,-30621        ; 8863h
88AB392C  addi r3,r10,30196     ; 75F4h
88AB3930  lwz r4,0(r11)
88AB3934  bl Msg (88ce8fdch)
}
88AB3938  addi r1,r1,96     ; 60h
88AB393C  lwz r12,-8(r1)
88AB3940  mtlr r12
88AB3944  blr
```

Registers

```
 r0 = 0000000000000000  r1 = 000000007005EB90  r2 = 0000000020000000
r3 = FFFFFFFF886375F4  r4 = 0000000000000000  r5 = 000000000000000A
r6 = 0000000000000000  r7 = 0000000000000000  r8 = 00000000887212F0
r9 = 0000000019999999 r10 = FFFFFFFF88632C20 r11 = 000000007FFFFFFF
r12 = 0000000088AB3924 r13 = 00000000801F0000 r14 = FFFFFFFF831E0000
r15 = FFFFFFFF82FAED7C r16 = 0000000000000001 r17 = 0000000000000000
r18 = 0000000000000000 r19 = 0000000000000000 r20 = FFFFFFFF82FAED7C
r21 = FFFFFFFF82B21128 r22 = FFFFFFFF831CCD0C r23 = FFFFFFFF83050000
r24 = FFFFFFFF831C5CA8 r25 = FFFFFFFF831C7D5C r26 = FFFFFFFF831C0000
r27 = 0000000000000000 r28 = FFFFFFFF82B21128 r29 = 0000000088DDC490
r30 = 0000000000000000 r31 = FFFFFFFF831C0000 cr = 24000028 xer = 00000000
```

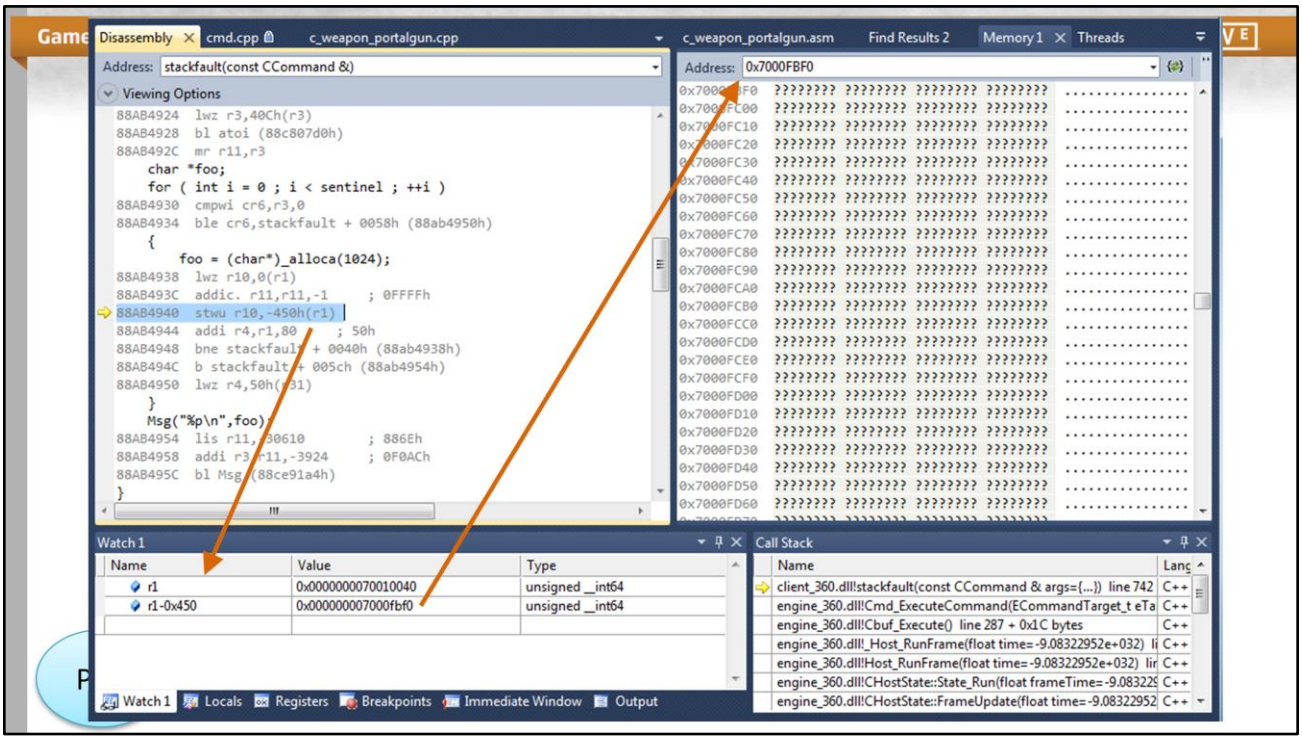
Locals | Registers | Breakpoints | Immediate Window | Output

Locals

Name	Value	Type
args	{m_nArgc=627310592 m_nArgv0Size=627449893 m_pArgSBuffer=0x886375fc "f %f" ...}	const CCommand

Locals | Registers | Breakpoints | Immediate Window | Output

Even if you didn't have that, it's still pretty easy to trace. You can see that the faulty instruction here is "load word from address in register 11." Well, register eleven contains an odd number, which can't possibly be word aligned, so boom.



Stack overflow. Well, this one's generally easy. Is it the "update stack" opcode? If so, is that stack pointer being updated to an address that isn't mapped memory?
Or, more generally, the CPU exception is "access violation." Is it trying to write to some pointer which is an offset from the stack pointer, and if so, is that offset in unmapped memory?

Jump through bad vtable

Call Stack

Name
61707320()
Client.dll!oops(const CCommand & args={...}) Line 730 + 0x
Client.dll!ConCommand::Dispatch(const CCommand & con
engine.dll!Cmd_ExecuteCommand(ECommandTarget_t eTar
engine.dll!Cbuf_Execute() Line 287 + 0x11 bytes

```
14E224A5  movss   xmm0,dword ptr [__real@3f800000 (
14E224AD  mov     esi,eax
14E224AF  mov     eax,dword ptr [esi]
14E224B1  mov     edx,dword ptr [eax+10h]
14E224B4  add     esp,4
14E224B7  mov     ecx,esi
14E224B9  movss   dword ptr [esp],xmm0
14E224BE  call    edx
14E224C0  mov     eax,dword ptr [esi]
14E224C2  movss   xmm0,dword ptr [__real@3f800000 (
```

```
    virtual float Slope( float t );
    virtu
    // ..
};

void oop
{
    Polyn
    Msg("
    delete foo;
}
```

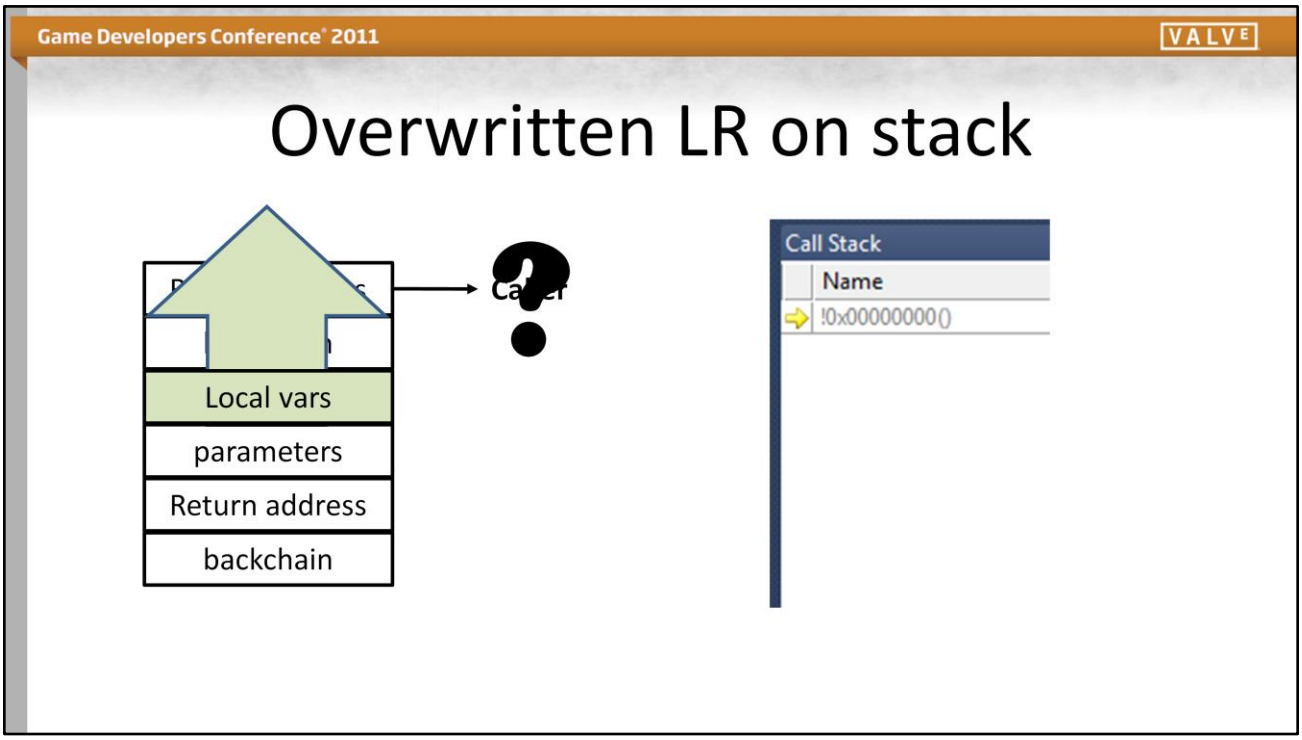
```
class CThingy
{
public:
    CThingy( int x ) : m_n(x) {}
    virtual int Get() { return m_n; }
    int m_n;
};
```

	Type
bcc8 {m_n=0x00000000 }	Polynomial *
0x00000000 }	CThingy
1018 const CThingy::`vftable'	*
0000	float *
m_degree 0x0zb40036	unsigned int

Okay, how about a jump through a bad virtual function.

Let's say you have a basic Polynomial class with six virtual functions, which means somewhere in memory is a table with six function pointers in it – its vtable. Then you have your function that calls a couple virtuals on it, and you blow up.

Jumping through a bad virtual function pointer will usually have a pretty obvious signature like this, where you jumped to an address that wasn't actually in the code segment – the call stack will contain a frame that's just a number, not corresponding to any legitimate function. But with a little investigation you can usually divine a little more about it. If you look at the code around the last frame on the stack, you can find that the Polynomial, which was being passed as ecx (the "this") pointer to the virtual function, was on esi. So we put esi in our watch window and AHA! Its vtable is not that of a Polynomial at all. It is that of a Thingy, which has only one virtual function. So, our factory function must have returned a pointer to something that wasn't actually a Polynomial.



Bad return addresses – aka the smashed stack. Insidious, annoying, and common. When one function calls another, of course it must store its return address somewhere in memory, usually on the stack. Well, great, except our local data's also on the stack, and thus when you write off the end of an array you can easily overwrite the backchain (making it impossible for the debugger to find parent stack frames) and the return address.

When this happens a function will "return" to an address that's garbage or NULL.

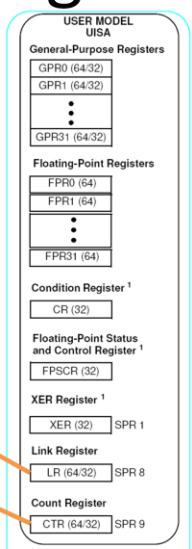
> MSVC will usually refuse to give you any kind of call stack when this happens, or even give you a memory window unless you switch to a different thread.

Jumps to 0x0000: quick triage

```
0:006> .ecxr
 r0=00000000`85617a04  r1=00000000`701cf920  r2=00000000`20000000
  ...
r30=00000000`0000000b r31=00000000`00000086 ctr=00000000`00000000
iar=00000000 msr=0000b030 lr=00000000 cr=22000042
```

- ctr = 0 : jump through bad vtable / func pointer
- lr = 0 : overwritten stack

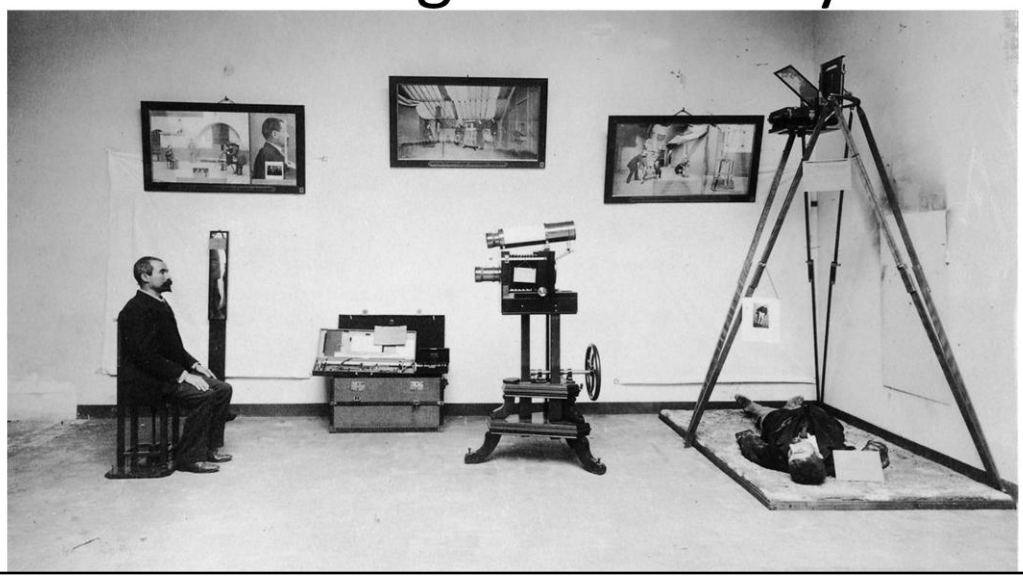
blr : "branch to link register"
return from function
bctr : "branch to count register"
call function pointer/virtual



PPC

There's exactly two opcodes on the PPC that provide a "jump to address", each of them using a specific register. Looking at which register caused the fault can give you a quick hint as to what went wrong. If it's the count register (used for function pointer calls), then you probably have a bad vpointer, or the like. If the Link Register is NULL, then something overwrite its cell on your stack.

Thinking Forensically



Ultimately you need to think like a detective at an accident scene. Here you have all these smashed bits and pieces of evidence, and you're trying to work backwards to figure out what caused them to get there like that. And sometimes it can take a little bit of detective work and inference to get to your conclusion.

(I'm not quite sure what that picture is… I typed "forensics" into the NIH's website and that came up.)

Stack Reconstruction

```
0x7005D8C0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x7005D8E0  [illegible] 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x7005D900  7005d960 [illegible]05dbc0 00000000 00000000 7005da80 8281c694 827e5634 dda199a0
0x7005D920  [illegible] 00000000 00000001 7005dac0 00000000 0000002f 00000002 7005da90
0x7005D940  [illegible]7005da10 00000000 00000000 00000000 dd328c30 00000000 00000001
0x7005D960  00000000 [illegible]82eb8 87376b54 00000000 00000000 00000000 ffffffff 831e0000
0x7005D980  [illegible] 831e0000 00000000 dd500640 00000000 00000001 00000000 00000000
0x7005D9A0  00000000 00000000 00000000 00000001 00000000 dc37e020 82225a78 00000001
0x7005D9C0  7005da20 dcb82eb8 855fbc9c 00000000 7005da40 84e0bea8 84d7596c 04fd0040
0x7005D9E0  7005da40 0000003d 00000000 00000000 00000000 dd7bb000 87381880 000700ff
0x7005DA00  7005daa0 000000fe 00000000 8281bdc0 00000000 dc37e020 82226128 0000002c
0x7005DA20  7005da90 dd328c30 00000000 00000000 7005da90 dd7bb000 87376b54 5f696e6e
0x7005DA40  7005daa0 00000000 00000002 7005dab0 00000000 dd67e780 00000000 00000000
0x7005DA60  ffffffff 86ba0000 00000000 00000000 7005daf0 84e0dfc8 00000000 00000000
0x7005DA80  6d6f6465 dc878848 82225e50 00000000 7005db10 dd7bb000 86bfc9d0 5f646573
0x7005DAA0  7005db50 74696f6e 2f727562 626c655f 77616c6c 5f636569 6c696e67 30310000
0x7005DAC0  6d6f6465 6c735c70 00000000 00000001 00000000 0000001c 00000000 00000001
0x7005DAE0  00000000 dca6db1c ffffffff ffffffff 00000000 00000079 00000000 00000001
```

Sometimes when part of your stack has been trashed, but you still have an r1 pointer and know generally where the bottom frame ought to be, you can make an educated guess about what the actual stack was by looking for consistent chains. Because the stack is so heavily trafficked, old stack frames – those “left over” as traces in memory from previous operations – would tend not to be internally consistent; if you have a chain that forms a complete, consistent link all the way back to main(), it has good odds of being the actual one under execution at the time of death.

Let’s say this is a stack, and your r1 is 0x7005D8C0. The bottom frame is all zeroed out, but the damage doesn’t seem to go too far. Maybe we can recover something. Any cell containing a number like 0x7005D*** is probably a pointer to a stack location, since it’s so near the value of r1 and we know a priori that’s where the stack is.

So, if we were to guess at `0x7005D900 being the bottom of a frame, that means that it links to 7005d960, and... No, that‘s null.`

Stack Reconstruction

```
0x7005D8C0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x7005D8E0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x7005D900  7005d960 7005dbc0 00000000 00000000 7005da80 31c694 827e5634 dda199a0
0x7005D920  7005d9c0 00000000 00000001 7005dac0 00000000 0000002f 00000002 7005da90
0x7005D940  00000002 7005da10 00000000 00000000 00000000 dd328c30 00000000 00000001
0x7005D960  00000000 dcb82eb8 87376b54 00000000 00000000 00000000 ffffffff 831e0000
0x7005D980  ffffffff 831e0000 00000000 dd500640 00000000 00000001 00000000 00000000
0x7005D9A0  00000000 00000000 00000000 00000001 00000000 dc37e020 82225a78 00000001
0x7005D9C0  7005da20 dcb82eb8 855fbc9c 00000000 7005da40 84e0bea8 84d7596c 04fd0040
0x7005D9E0  7005da40 0000003d 00000000 00000000 00000000 dd7bb000 87381880 000700ff
0x7005DA00  7005daa0 000000fe 00000000 8281bdc0 00000000 dc37e020 82226128 0000002c
0x7005DA20  7005da90 dd328c30 00000000 00000000 7005da90 dd7bb000 87376b54 5f696e6e
0x7005DA40  7005daa0 00000000 00000002 7005dab0 00000000 dd67e780 00000000 00000000
0x7005DA60  [illegible]6ba0000 00000000 00000000 7005daf0 84e0dfc8 00000000 00000000
0x7005DA[illegible] 6d6f6465 [illegible]78848 82225e50 00000000 7005db10 dd7bb000 86bfc9d0 5f646573
0x7005DAA0  [illegible]4696f6e 2f727562 626c655f 77616c6c 5f636569 6c696e67 30310000
0x7005DAC0  6d6f6465 6c735c70 00000000 00000001 00000000 0000001c 00000000 00000001
0x7005DAE0  00000000 dca6db1c ffffffff ffffffff 00000000 00000079 00000000 00000001
```

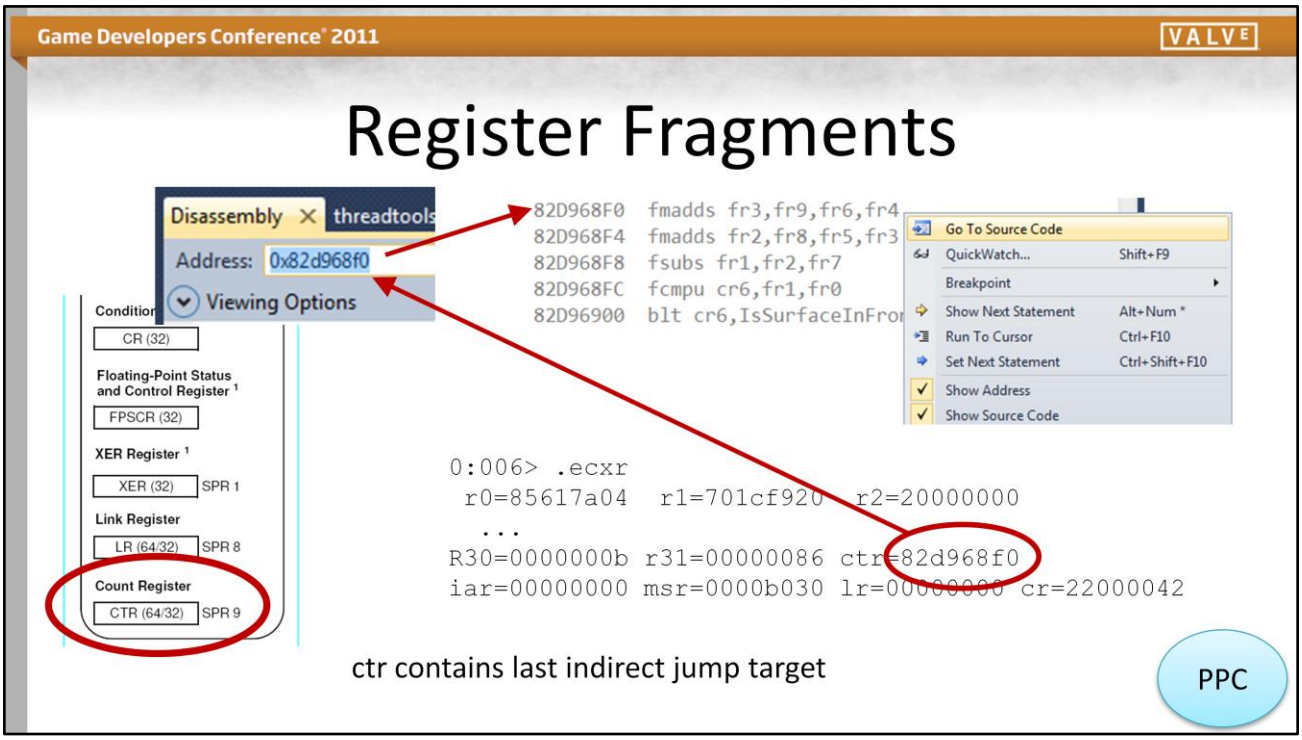
How about this one? No, that's not a legitimate stack address.

Stack Reconstruction

```
0x7005D8C0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x7005D8E0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x7005D900  [illegible]005dbc0 00000000 00000000 7005da80 8281c694 827e5634 dda199a0
0x7005D9[illegible]  7005d9c0  [illegible]00000 00000001 7005dac0 00000000 0000002f 00000002 7005da90
0x7005D940  [illegible]005da10 00000000 00000000 00000000 dd328c30 00000000 00000001
0x7005D960  00000000 dcb82eb8 87376b54 00000000 00000000 00000000 ffffffff 831e0000
0x7005D980  ffffffff 831e0000 00000000 dd500640 00000000 00000001 00000000 00000000
0x7005D9A0  [illegible]00000000 00000000 00000001 00000000 dc37e020 82225a78 00000001
0x7005D9[illegible]  7005da20  [illegible]82eb8 855fbc9c 00000000 7005da40 84e0bea8 84d7596c 04fd0040
0x7005D9E0  [illegible]000003d 00000000 00000000 00000000 dd7bb000 87381880 000700ff
0x7005DA00  [illegible]00000fe 00000000 8281bdc0 00000000 dc37e020 82226128 0000002c
0x7005D[illegible]  7005da90  [illegible]28c30 00000000 00000000 7005da90 dd7bb000 87376b54 5f696e6e
0x7005DA40  [illegible]00000000 00000002 7005dab0 00000000 dd67e780 00000000 00000000
0x7005DA60  ffffffff 86ba0000 00000000 00000[illegible]4e0dfc8 00000000 00000000
0x7005DA80  6d6f6465 dc878848 82225e50 00000  7005db10  [illegible]bb000 86bfc9d0 5f646573
0x7005DAA0  7005db50 74696f6e 2f727562 626c655[illegible]1636569 6c696e67 30310000
0x7005DAC0  6d6f6465 6c735c70 00000000 00000001 00000000 0000001c 00000000 00000001
0x7005DAE0  00000000 dca6db1c ffffffff fff[illegible]000079 00000000 00000001
```

How about d920? Well, that links to d9c0, to da20, and that looks like a consistent chain! And indeed if you follow that up you get back to main, so that looks real.

And of course you can write debugger scripts to do this for you, based on a guess at a stack address.



Sometimes the registers contain trace evidence that can point near the fault. For example, remember that ctr contains the address of the last virtual function called. So, if your crash is due to a bad LR, CTR might still contain a pointer to the most recently called virtual function, perhaps even the function that crashed. Even if not, it might be some function called recently, which can be an important clue.

Working Backwards: Case Study

BUG #84092: CRASH starting game.

"Happened only on the kits set to Swedish and Danish."

\\dumpserver\Test26\undescriptivename.dmp
\\dumpserver\Test27\lessdescriptivename.dmp

Now let me take you through a fun one that actually came up while I was putting these slides together.

Callstack [PPU]

Type	Function
	??? 0x00000000
	??? 0xDDDDDDDD
	??? 0xDDDDDDDD
	??? 0xDDDDDDDD

```
r0=0000000000000000
r1=00000000D00FCC50
r2=000000000107EA10
r3=FFFFFFFFFFFFFFFF
r4=FFFFFFFFFFFFFFFF
r5=00000000D00FCC99
r6=0000000000000000
r7=0000000000000020

ctr=000000000025AEA4
lr=0000000000000000
pc=0000000000000000
```

void CStdMemAlloc::Free(void *pMem)

PPC

I opened up one of those dumps and got this call stack, so I knew that a) something had trashed my stack and b) I wasn't going to have a good day. The first thing I looked at was the registers pane, and I saw that LR was null, but CTR had a valid code address, so I looked at that, and found it was pointing to… free(). Okay, that's not very specific at all.

Callstack [PPU]

Type	Function
	??? 0x00000000
	??? 0xDDDDDDDD
	??? 0xDDDDDDDD
	??? 0xDDDDDDDD

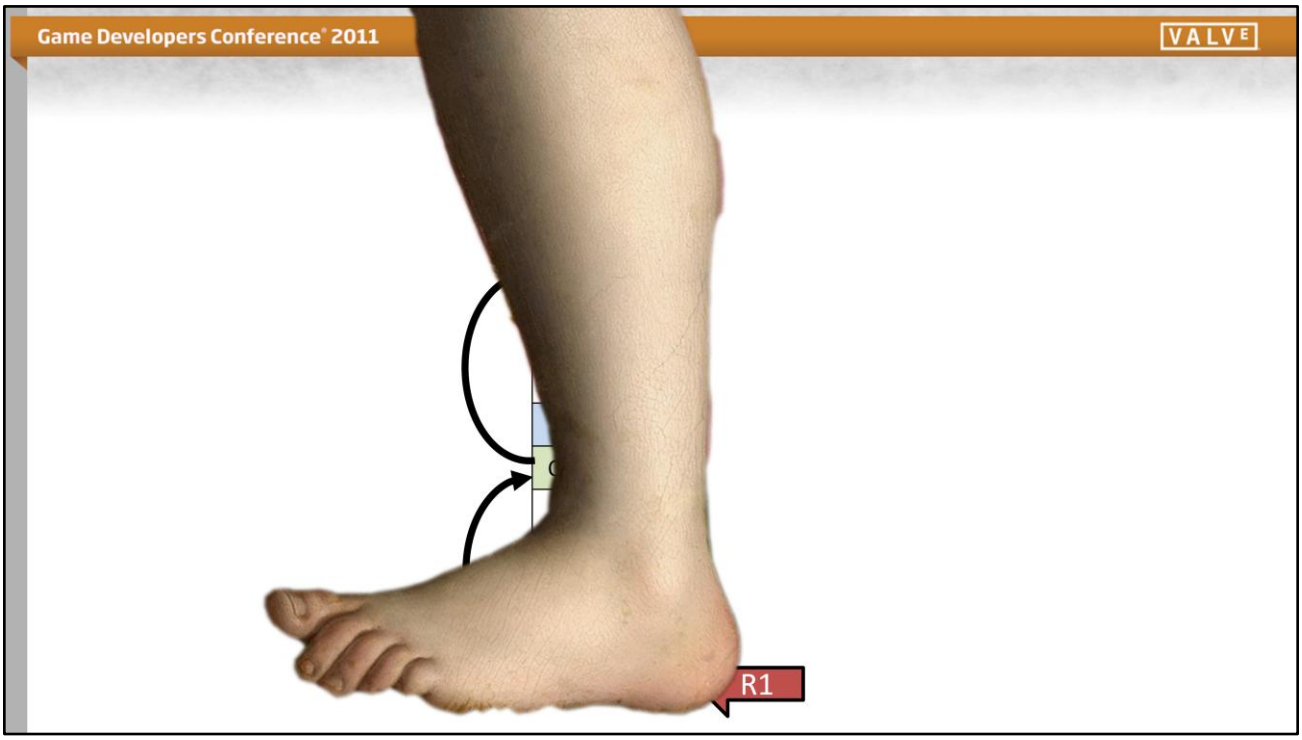
```
r0=0000000000000000
r1=00000000D00FCC50
r2=000000000107EA10
r3=FFFFFFFFFFFFFFFF
r4=FFFFFFFFFFFFFFFF
r5=00000000D00FCC99
r6=0000000000000000
r7=0000000000000020

ctr=000000000025AEA4
lr=0000000000000000
pc=0000000000000000
```

Memory - 4 Byte Hex [PPU]

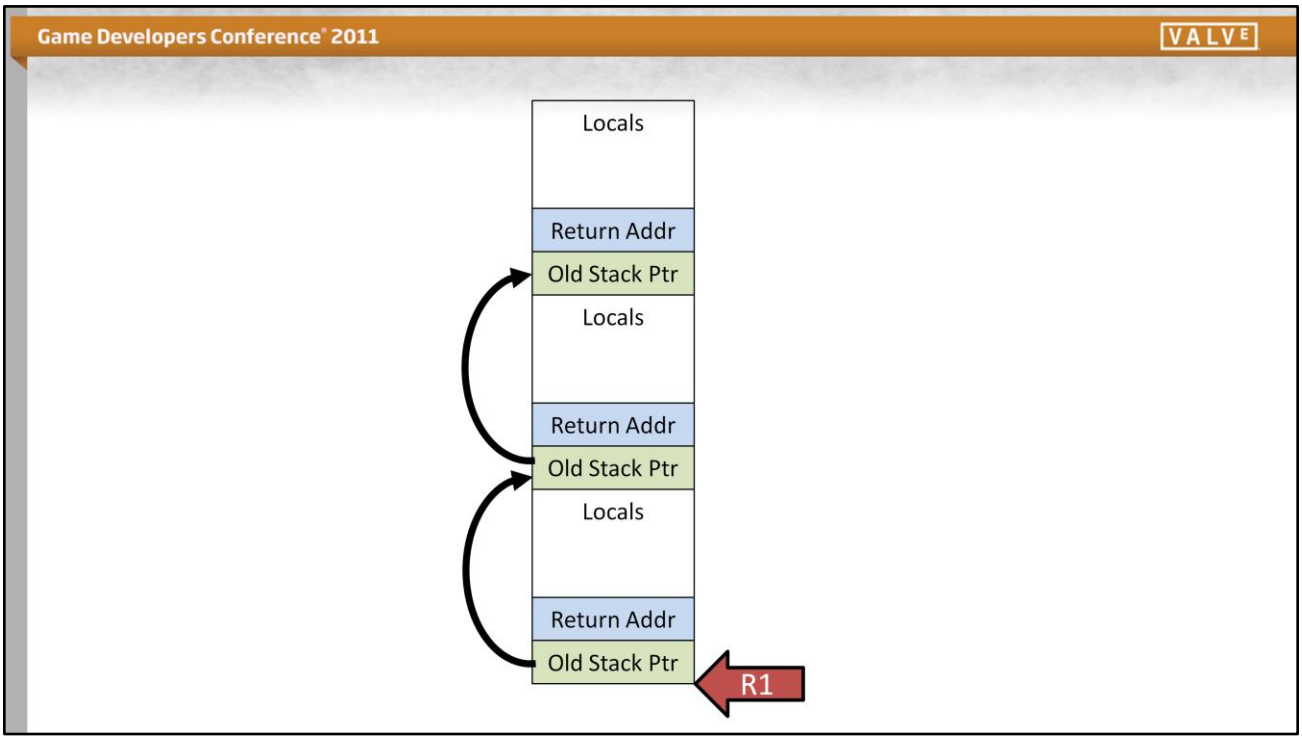
Address r1 Columns 4

```
D00FCC50  FFFFFF22 00000000 00000000 00000000
D00FCC60  00000000 00000000 00000000 00000000
D00FCC70  00000000 00000000 00000000 00000000
D00FCC80  00000000 00000000 FFFFFF30 FFFFFFFE
D00FCC90  FFFFFF9E FFFFFF1C FFFFFF3E FFFFFFE4
D00FCCA0  FFFFFFC3 00000000 00000000 00000000
D00FCCB0  00000000 00000000 00000000 00000000
D00FCCC0  00000000 00000000 00000000 00000000
D00FCCD0  00000000 00000000 FFFFFF7B FFFFFFF8
D00FCCE0  FFFFFF06 00000000 00000000 FFFFFF68
D00FCCF0  FFFFFFFF FFFFFF13 00000000 00000000
D00FCD00  00000000 00000000 00000000 00000000
D00FCD10  00000000 00000000 00000000 00000000
D00FCD20  00000000 00000000 FFFFFF75 FFFFFFFB
D00FCD30  FFFFFF0D 00000000 00000000 FFFFFF73
D00FCD40  FFFFFFFD FFFFFF0E 00000000 00000000
D00FCD50  00000000 00000000 00000000 00000000
D00FCD60  00000000 00000000 00000000 00000000
```

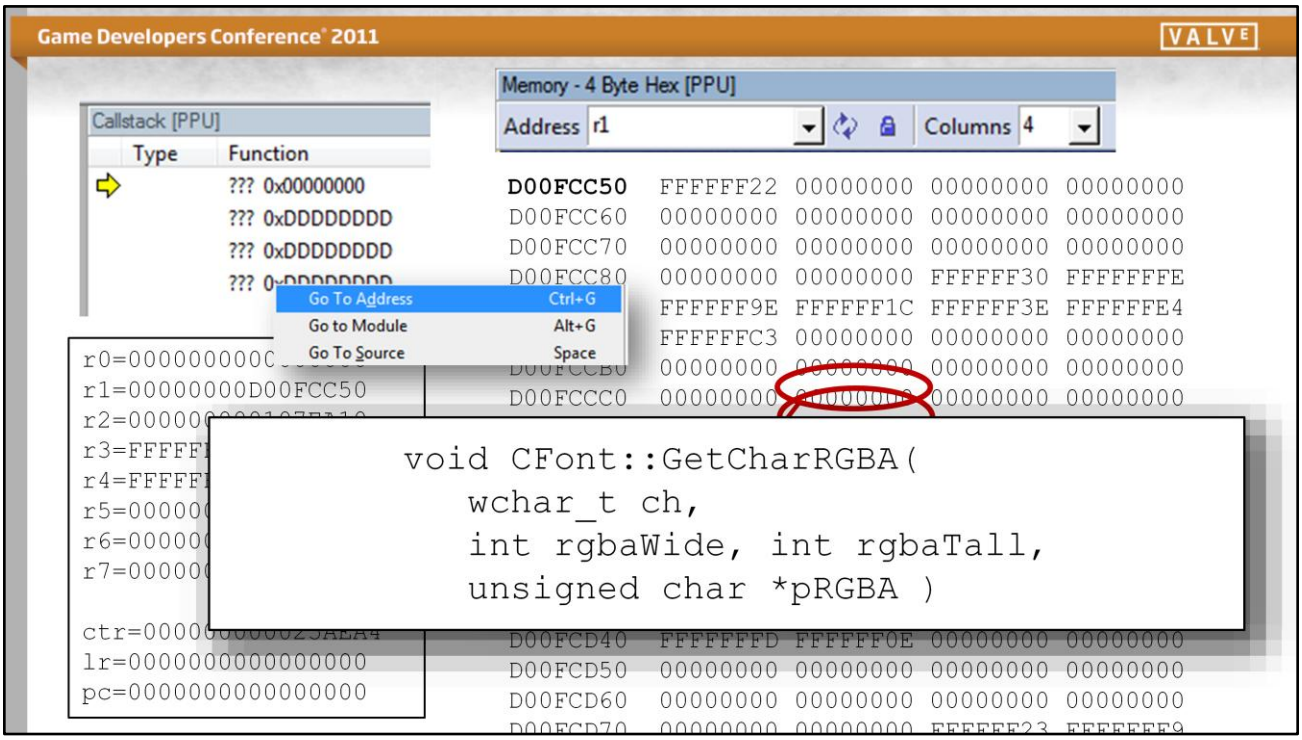
Well, r1 is still legit, so it points at the stack. What's on my stack frame? A bunch of zeroes and FFs, apparently, so I can see my stack has been well and truly trashed for quite a ways.



Remember that stack frames form a linked list in memory, where the “backchain” cell of each frame is the pointer to the previous frame. The way that the debugger reconstructs your call stack pane is by walking that linked list. Previously I showed how to do that by hand by looking upwards a few frames past the damage, but here it looks like the entire stack has been… stepped on.



Except when you “return” from a frame, you don’t actually delete anything. You just move the pointer up. The old frames, from the functions you called, stay there in memory like ghosts.

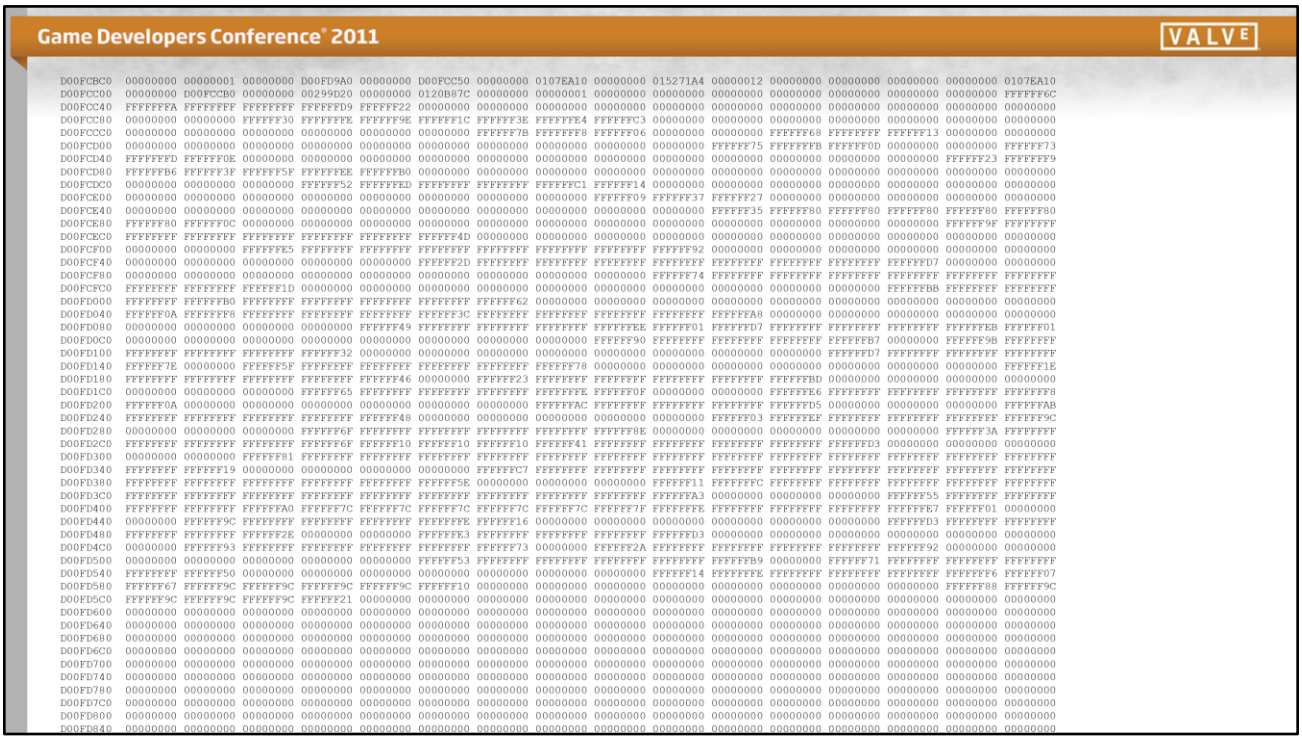


So, on the assumption that my function must have previously called some other function in the past, I looked *down* the stack – towards where a callee's frame would have been – to look for any frames that pointed at the *current* r1 in the exception. And there is one! Okay, if that's a backchain, then the old return address must be here, try looking that up in the code, and…

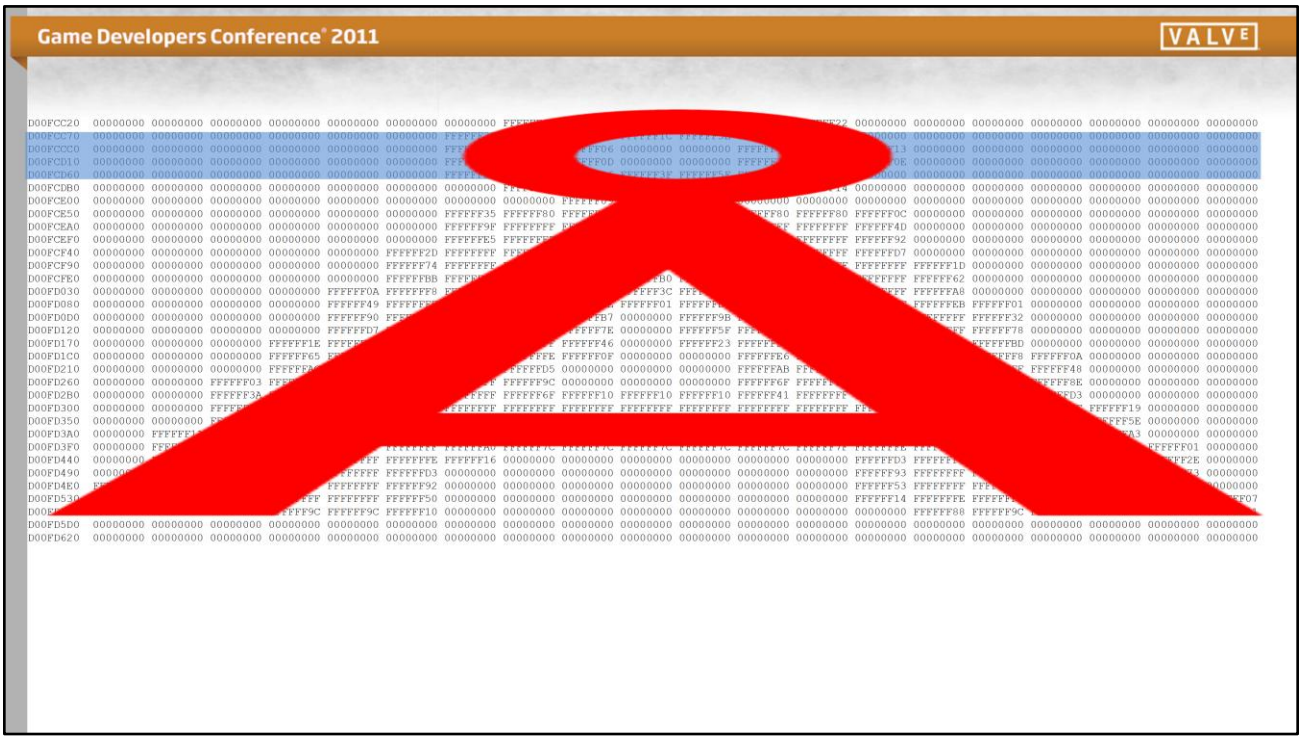
Boom, at some point, we called: void CFont::GetCharRGBA(wchar_t ch, int rgbaWide, int rgbaTall, unsigned char *pRGBA)

```
D00FCC00  00000000 D00FCCB0 00000000 00299D20 00000000 0120B87C 00000000 00000001
D00FCC20  00000000 00000000 00000000 00000000 00000000 00000000 00000000 FFFFFF6C
D00FCC40  FFFFFFFA FFFFFFFF FFFFFFFF FFFFFFD9 FFFFFF22 00000000 00000000 00000000
D00FCC60  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCC80  00000000 00000000 FFFFFF30 FFFFFFFE FFFFFF9E FFFFFF1C FFFFFF3E FFFFFFE4
D00FCCA0  FFFFFFC3 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCCC0  00000000 00000000 00000000 00000000 00000000 00000000 FFFFFF7B FFFFFFF8
D00FCCE0  FFFFFF06 00000000 00000000 FFFFFF68 FFFFFFFF FFFFFF13 00000000 00000000
D00FCD00  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCD20  00000000 00000000 FFFFFF75 FFFFFFFB FFFFFF0D 00000000 00000000 FFFFFF73
D00FCD40  FFFFFFFD FFFFFF0E 00000000 00000000 00000000 00000000 00000000 00000000
D00FCD60  00000000 00000000 00000000 00000000 00000000 00000000 FFFFFF23 FFFFFFF9
D00FCD80  FFFFFFB6 FFFFFF3F FFFFFF5F FFFFFFEE FFFFFFB0 00000000 00000000 00000000
D00FCDA0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCDC0  00000000 00000000 00000000 FFFFFF52 FFFFFFED FFFFFFFF FFFFFFFF FFFFFFC1
D00FCDE0  FFFFFF14 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCE00  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCE20  FFFFFF09 FFFFFF37 FFFFFF27 00000000 00000000 00000000 00000000 00000000
D00FCE40  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCE60  00000000 00000000 FFFFFF35 FFFFFF80 FFFFFF80 FFFFFF80 FFFFFF80 FFFFFF80
D00FCE80  FFFFFF80 FFFFFF0C 00000000 00000000 00000000 00000000 00000000 00000000
D00FCEA0  00000000 00000000 00000000 00000000 00000000 00000000 FFFFFF9F FFFFFFFF
D00FCEC0  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF4D 00000000 00000000
D00FCEE0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCF00  00000000 00000000 FFFFFFE5 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
D00FCF20  FFFFFFFF FFFFFF92 00000000 00000000 00000000 00000000 00000000 00000000
D00FCF40  00000000 00000000 00000000 00000000 00000000 FFFFFF2D FFFFFFFF FFFFFFFF
D00FCF60  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFD7 00000000 00000000
D00FCF80  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FCFA0  00000000 FFFFFF74 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
D00FCFC0  FFFFFFFF FFFFFFFF FFFFFF1D 00000000 00000000 00000000 00000000 00000000
D00FCFE0  00000000 00000000 00000000 00000000 00000000 FFFFFFBB FFFFFFFF FFFFFFFF
D00FD000  FFFFFFFF FFFFFFB0 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF62 00000000
D00FD020  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FD040  FFFFFF0A FFFFFFF8 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF3C FFFFFFFF FFFFFFFF
D00FD060  FFFFFFFF FFFFFFFF FFFFFFA8 00000000 00000000 00000000 00000000 00000000
D00FD080  00000000 00000000 00000000 00000000 FFFFFF49 FFFFFFFF FFFFFFFF FFFFFFFF
D00FD0A0  FFFFFFEE FFFFFF01 FFFFFFD7 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFEB FFFFFF01
D00FD0C0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
D00FD0E0  FFFFFF90 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFB7 00000000 FFFFFF9B FFFFFFFF
D00FD100  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF32 00000000 00000000 00000000 00000000
D00FD120  00000000 00000000 00000000 00000000 FFFFFFD7 FFFFFFFF FFFFFFFF FFFFFFFF
D00FD140  FFFFFF7E 00000000 FFFFFF5F FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF78
D00FD160  00000000 00000000 00000000 00000000 00000000 00000000 00000000 FFFFFF1E
D00FD180  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF46 00000000 FFFFFF23 FFFFFFFF
D00FD1A0  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFBD 00000000 00000000 00000000 00000000
D00FD1C0  00000000 00000000 00000000 FFFFFF65 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
D00FD1E0  FFFFFF0F 00000000 00000000 FFFFFFE6 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFF8
D00FD200  FFFFFF0A 00000000 00000000 00000000 00000000 00000000 00000000 FFFFFFAC
D00FD220  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFD5 00000000 00000000 00000000 FFFFFFAB
D00FD240  FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFF48 00000000 00000000 00000000
```

Armed with this knowledge, we can go back to the memory window and make an educated guess about the pattern there. Does it show up with eight column width?



Sixteen column?



Twenty column?

Do you see it?

ENHANCE!

And our stack frame – it's too small for you to see – is supposed to be there!

So what happened here is our font had bad metrics for certain diacritics, and in this case when we went to render the Swedish Å, the circle actually poked up out of its memory and into the stack frame!

COLLECTING EVIDENCE

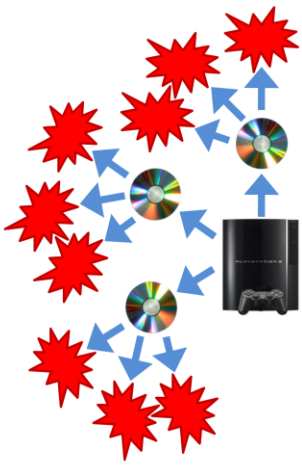
Making All The Bodies End Up In The Same Place

Returning to the subject of core dumps and getting them from QA. You want to have all your core dump files end up in a central location. Consider the alternative: let's say each tester, on encountering a crash, enters a bug. Then to that bug she attaches a dump file, like attaching it to an email. So you have a lot of bugs each with dumps in them. Okay, that's fine when you have one tester…

Ad Hoc Bugging

..playing one image…

Ad Hoc Bugging

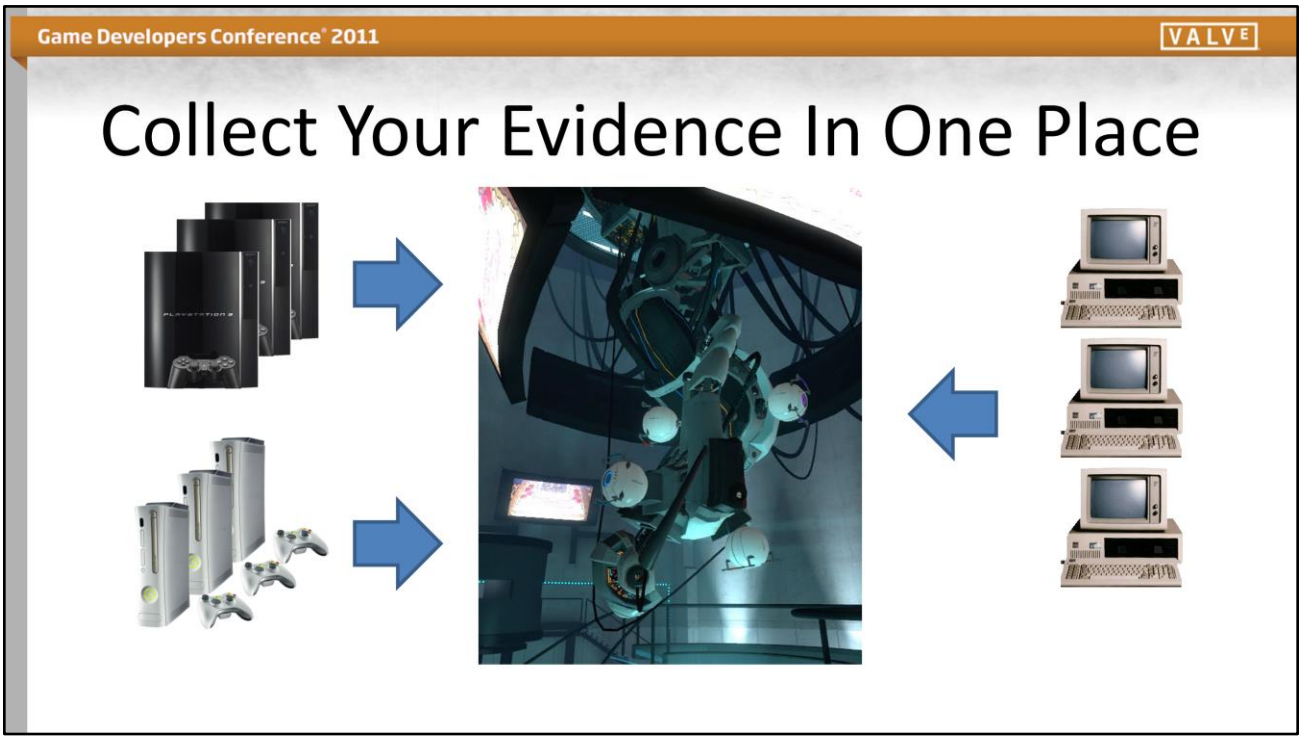


But really you have a fresh image every day.

And lots of testers on different platforms.

Lots and lots of different platforms. Maybe PC and Mac too.

Managing each crash individually like this is the road to madness.



So it can help a lot to configure each of your testing endpoints so that they automatically upload all of their dumps to a centralized place, or at least you have some automated process that sweeps them there. Once you've got that level of automation, you can add a bunch of other useful steps to it as well.

Review: Anatomy of a Core Dump

Usually

- Name/timestamp of app
- CPU context (registers)
- Exception data
- List of running threads
  - CPU context for each
  - Their callstacks
- List of loaded modules
- Memory address maps

Sometimes

- Entire contents of stack
- OS state
  - Names of other processes
  - Mutexes, etc
- Entire process memory ("full dump")
  - On 360, not "physically mapped" pages
- Custom additions

What's in a dump?

Exception data: what caused the crash. Was it a segfault, an invalid instruction?

On the 360, if you have any memory mapped via "physicalalloc", ie to get a 16mb page, it won't appear in a dump file even if you've configured for "full dump."

Symbols

- A symbol file maps source-level constructs to machine code and addresses in the executable.
- Produced by compiler/linker along with exe
- Debugger needs matching:
  - Source
  - Executable
  - Symbols
  - Data

Symbol files get invalidated whenever the output binary changes, so

Each platform has its own format of symbol file.

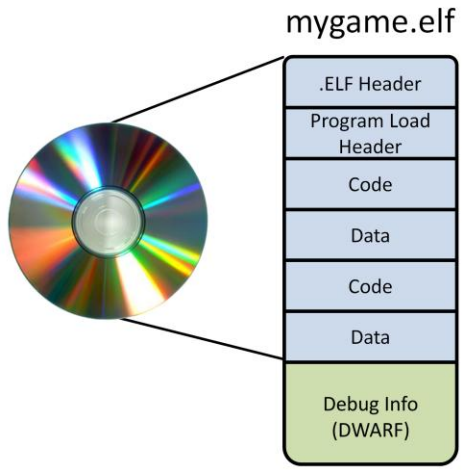
PDBs

- MSVC and other Microsoft tools
- Proprietary format
- Generated alongside .exe's and .dll's
- Symbols available even for "release" builds

Windows uses the Program DataBase format. The symbols go into separate PDB files that are generated alongside the main executable dlls. One consequence of this is that the "release" configuration doesn't omit symbols from the executable – you always have symbols available for anything you build, so long as you keep the PDBs around. The difficulty of optimizing release Windows builds is mostly due to optimizer rearrangement.

DWARF format

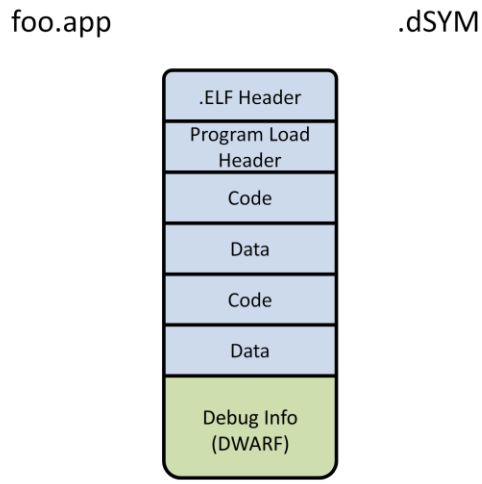
- GCC derivatives (Linux & PS3)
- Symbol data is part of the .elf
- Debug data "stripped" from retail elves



http://dwarfstd.org/

.dsym

- Apple executables contain symbol data
- Symbols can be omitted, or stripped into a separate file



On Mac, symbol info also gets cooked into the executable, but dsymutil lets you strip it out into its own file. Also, most released executables have some light symbol info, enough to get the function names for a call stack, though not source line numbers.

Making a Symbol Server

- Changing the binary changes symbols
- ∀ crash, you need the exact same version of:
  - Exe that made it
  - Symbols
  - Source code
- And a way to figure out where to get the info corresponding to a given dump!

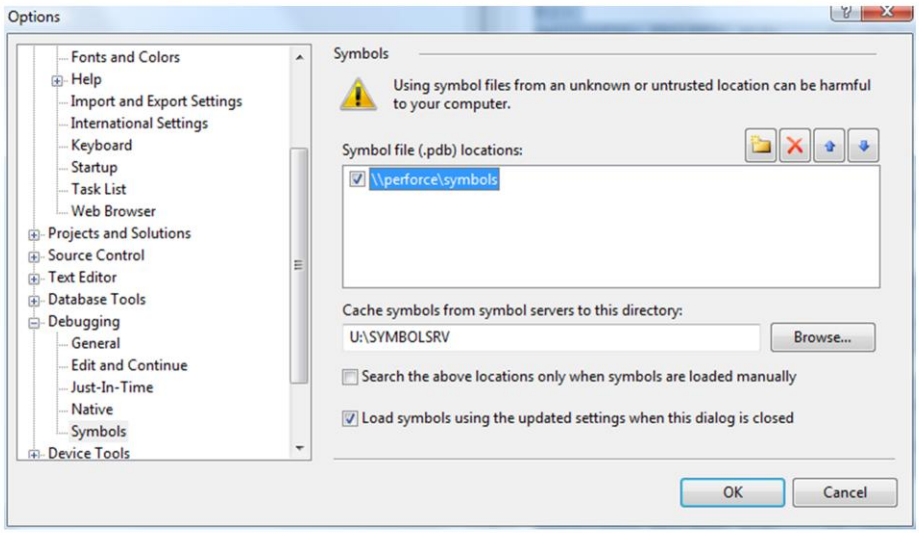
Making a Symbol Server (PC/360)

- .exe/.dlls get signatures from timestamp&size
- which identify PDBs for a given source .exe
- A PDB symbol server is just an ordinary Windows fileshare
- Use symstore.exe to make one

Windows/360

MSFT documentation for symbol store:
http://msdn.microsoft.com/en-us/library/ms681417(VS.85).aspx

Matching Dumps with Symbols (MSVC)



MSVC can connect a dump with its symbols so long as you tell it where to look for the symbol database.

Making a Symbol Server (PS3)

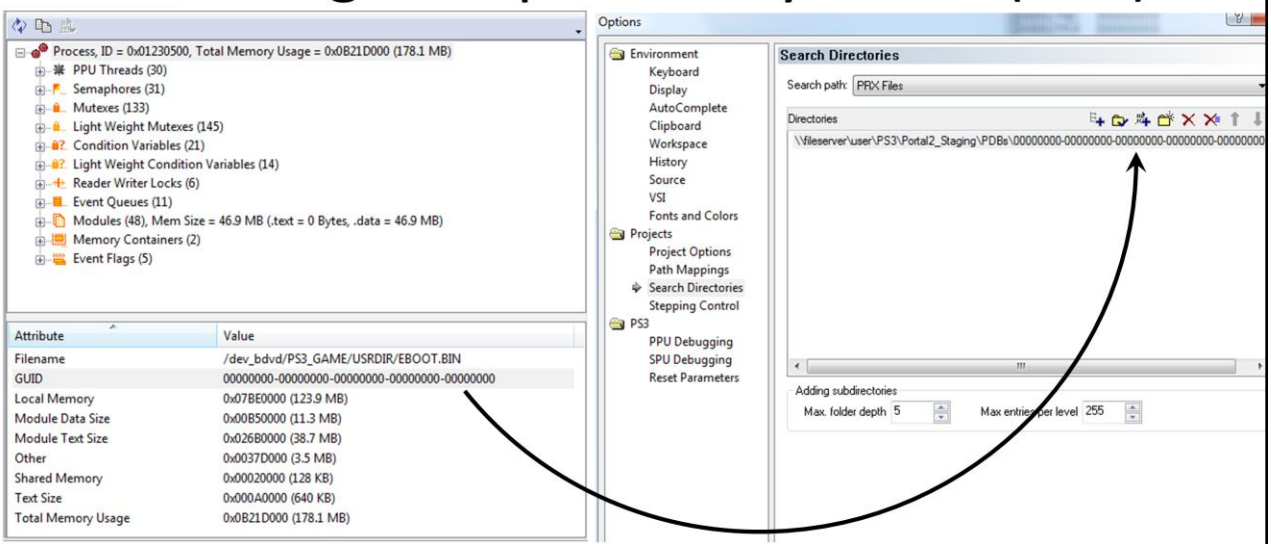
- You have to roll your own. Here's how we did.
- Recent compiler versions can cook a GUID into the .elf/prx by hashing the image:

```
u:\image\PS3_GAME\USRDIR>ps3bin --dump-sections=.PpuGUID mygame.self
14 - .PpuGUID:
  Type:        SHT_PROGBITS (0x00000001)
  Flags:       SHF_ALLOC
  Address:     0x00000000000AED50 | Offset:      0x000000000009ED50
  Size:        0x0000000000000020 | Link:        0x00000000
  Info:        0x00000000         | Align:       0x0000000000000008
  Entry Size:  0x0000000000000000

0x000AED50  7F 47 55 49 44 01 00 00 0A DB 4C 68 40 77 87 91  .GUID.....Lh@w..
0x000AED60  E1 CA 15 D1 01 3F 89 A2 B1 B3 86 6A 00 00 00 00  .....?.....j....

u:\image\PS3_GAME\USRDIR>copy mygame.self
     \\symbolserver\mygame\0ADB4C68-40778791-E1CA15D1-013F89A2-B1B3866A\mygame.self
```

Doing the same on the PS3 is going to require a little more manual effort. The compiler and linker can cook a GUID into each elf and prx by hashing the binary image (which means the GUID will be the same if you rebuild the image from the same source). You can extract that GUID from the binary with something like ps3bin.exe, and then store the .self, which contains the symbols, in some kind of database.

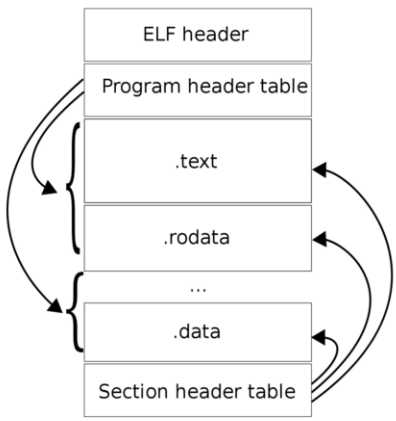
I find a simple filesystem works pretty well as a database – just put each file under a directory named after the GUID.

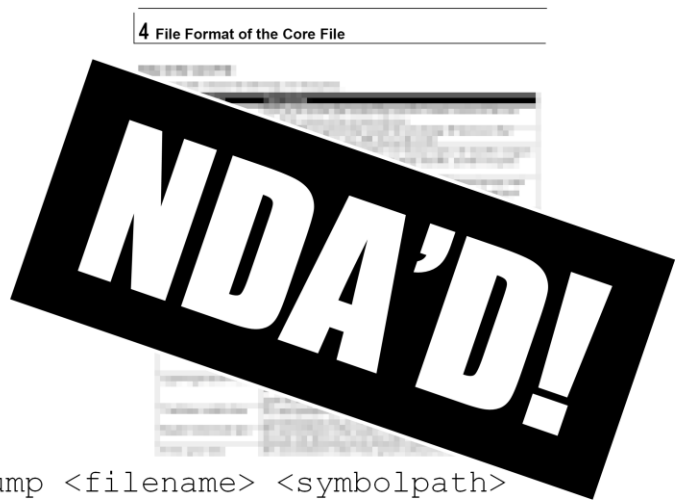
Matching Dumps with Symbols (PS3)



Once again, it's a little harder with Sony. A crash dump will contain the GUID of the executable that emitted it, but getting that info is tricky. You can load your dump in the Target Manager or the debugger and then look at the Process section of the kernel pane; that will contain the GUID, which you can then use to find wherever you put your symbol files on a server.

Matching Dumps with Symbols (PS3)






```
ps3debugger.exe -f -coredump <filename> <symbolpath>
```

The alternative is to use the fact that dump files are also .elfs and Sony's documentation of the core file format to go digging through the dump for the GUID info. Once you have the GUID, again you use that to find your symbols in your server and launch the debugger appropriately.

Automated Dump Collection

As mentioned before, it's easiest if all of your minidump files end up in the same place automatically. This isn't hard; you can either configure your kits to all write their dump files to a shared file server, or you can have each PC and kit emit the dumps to their local hard drives, and then you write some scheduled task or other robot that sweeps them over to your central location. You can get fancy with databases if you like.

Automated Dump Triage

- Preliminary, basic info without having to open a debugger
  - *ie,* Call stack, exception type, registers in a .txt file
- Sorting and grouping crashes by cause
- Overview statistics to set priority and spot trends

See also…

"Debugging in the (Very) Large: Ten Years of Implementation and Experience"
Microsoft: Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt

What works for one dump doesn't work for one hundred – as the incoming storm scales up, you won't have time to open each dump individually in the debugger. Automating part of the process will free up more time to investigate each issue, and psychologically, the easier it is to deal with a stability problem, the more likely a programmer is to deal with it.

One good place to start is to rig a script to automatically get some basic information on every dump, like exception type and a call stack, and put that into a .txt file next to the dump in your repository; thus you can get basic info on each dump in five seconds instead of waiting for ProDG to spool up.

These are individual steps you can take incrementally as your influx scales up – the most important is the at-a-glance readout, then bucketing, etc.

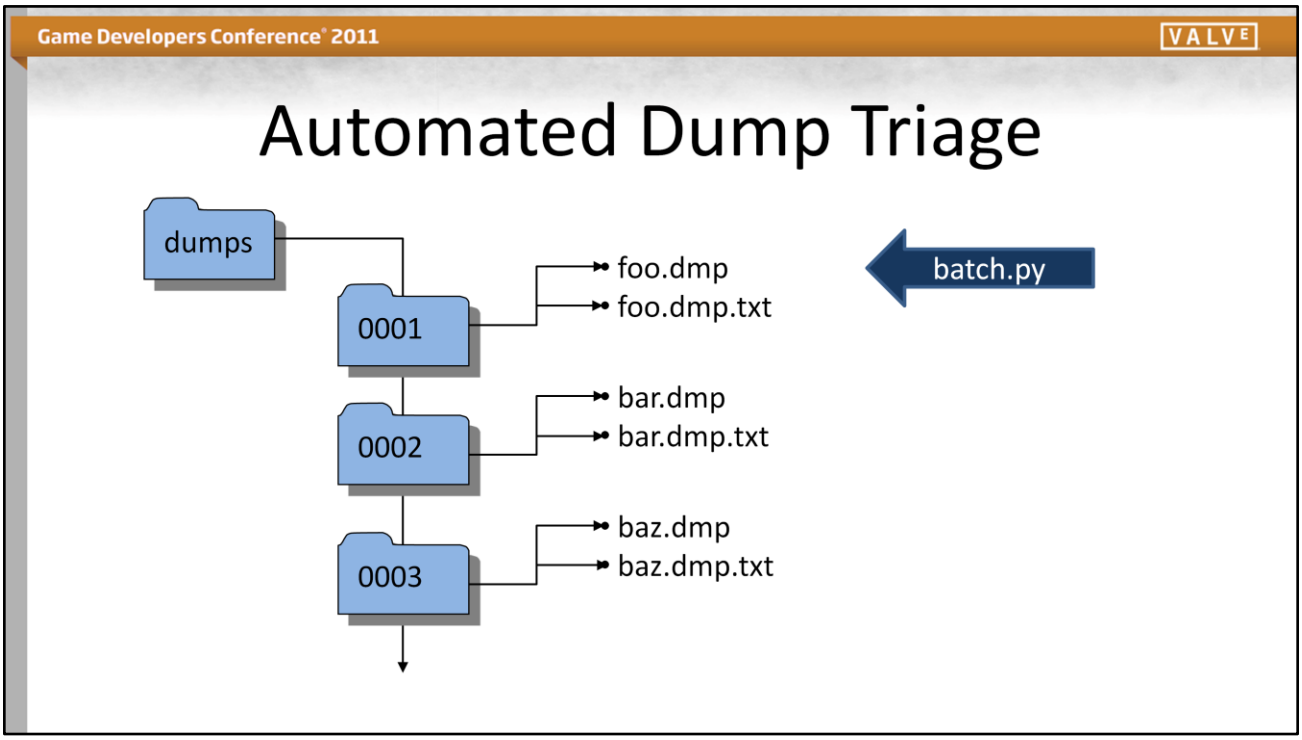
Automated Dump Triage

- MSVC (Windows/360): WinDbg/`kd` (command-line debuggers)

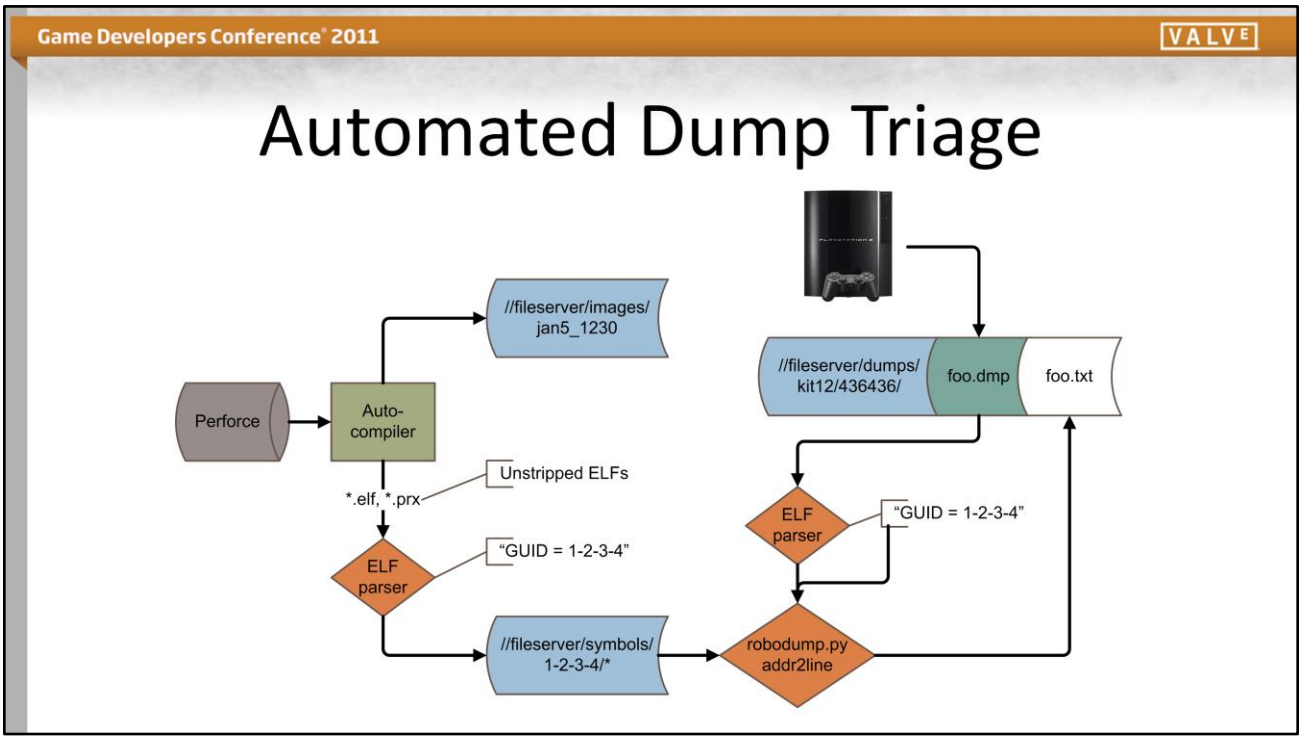
```
> i386kd.exe -lines -cf scriptfile -z core dump >dumpinfo.txt
```

`.sympath+ srv*\\myserver\symbols`	*add symbol path*
`!reload`	*reload symbols*
`.ecxr`	*print exception record*
`r`	*print registers*
`kb`	*print call stack*
`lm lv`	*print loaded modules*
`q`	*quit*

Windbg for robo-cracking on 360/windows



Then you can make a scheduled batch file out of this and it can just march through your dump repository generating its report for each one in turn.



On the PS3 again we had to build our own tools. Let's go through the whole chain. Our autobuilder pulls a changelist from perforce and starts building an image from it. Once the executables have been compiled, a script parses through the .ELF for the GUID the linker cooked in there, and then creates a directory on a central fileserver named with that guid. It copies the entire .elf and .prx tree (because the executables *are* the symbols in the .ELF format) onto the fileserver. Later, when a test kit emits a dump, we copy that onto the fileserver as well. We use the ELF parser on the dump file to find the GUID, which gives us the matching directory on the fileserver. Once we have that, I manually walk the stack found in the dump file, correlate each address on the stack against the symbols to come up with a function name, source file, and line number, then emit an exception report onto the fileserver adjacent to the dump.

Evolve Psychic Powers

```
analyze_all.py > todaysdumps.txt

outlook.exe /c ipm.note
  /m myself@example.com /a todaysdumps.txt
```

This is Arthur Fellig, a famous news photographer of the early 20th century. He was better known under the name Weegee (Oujia), because of his preternatural ability to show up at crime scenes before the police did. (He had one of the first police radio scanners.)

So here is my Magic Batch File that emailed me the stack trace for every 360 crash during Left4Dead QA in real time

hint: you can use this to pretend you are psychic and know about bugs before they are submitted

Always Do The Autopsy

- Don't assume causes from symptoms
- At least eyeball the call stacks.

When in doubt about a crash, do the autopsy. You might find a surprising cause, or reveal a core bug that hasn't shown elsewhere.

A THOUSAND POINTS OF DATA

Bugs Don’t Stop After Shipping

Gathering Stability Data From Customers

- You can't catch everything in QA
  - e^n user configurations out there
  - 3^d parties break your game
- Customers = 1,000,000 testers
- Built into OS:
  - Windows: WinQual, emailing .mdmps
  - Mac: Crash Report dialog, "please copy paste"

You can't catch every issue in testing, especially on PC

umpty-bazillion possible end user configurations means someone out there is going to hit an edge case you didn't think of

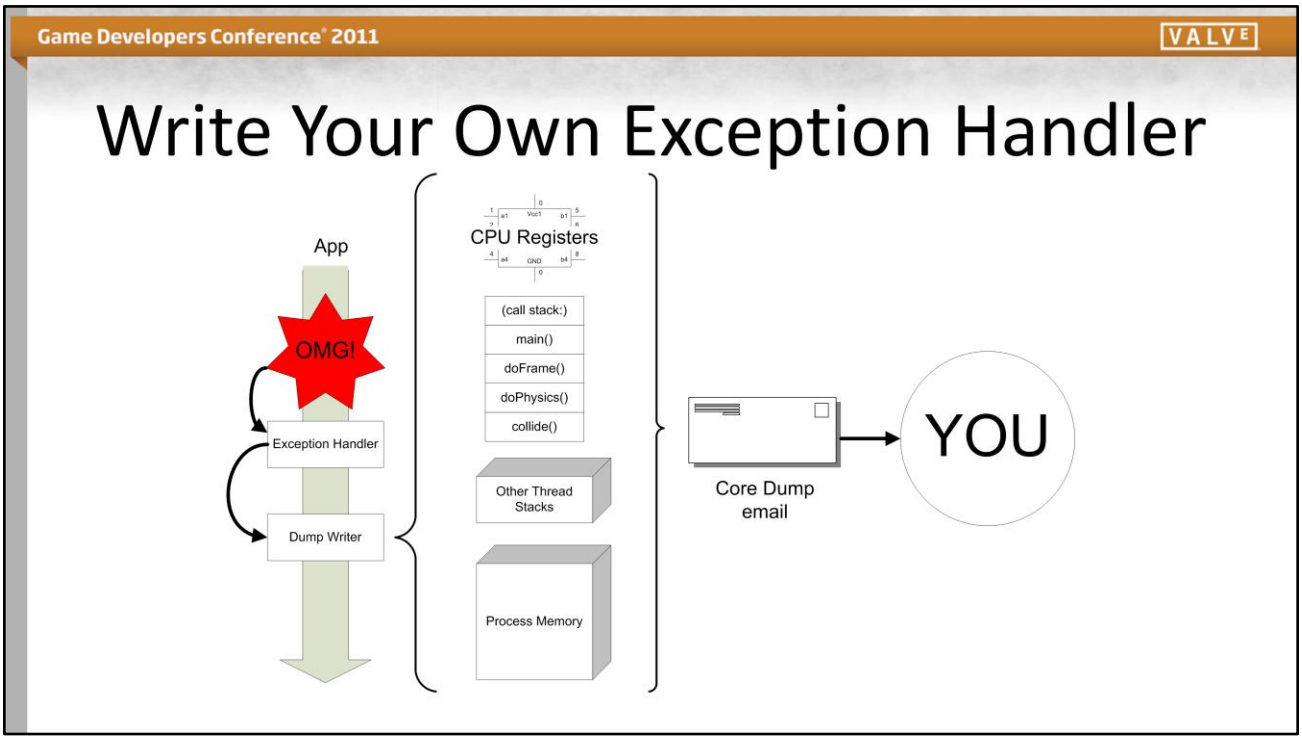
third party software updates can break your game (ie graphics card drivers) via incompatibility

By accumulating data from your customers, you get millions of testers for free and can quickly roll that into updates.

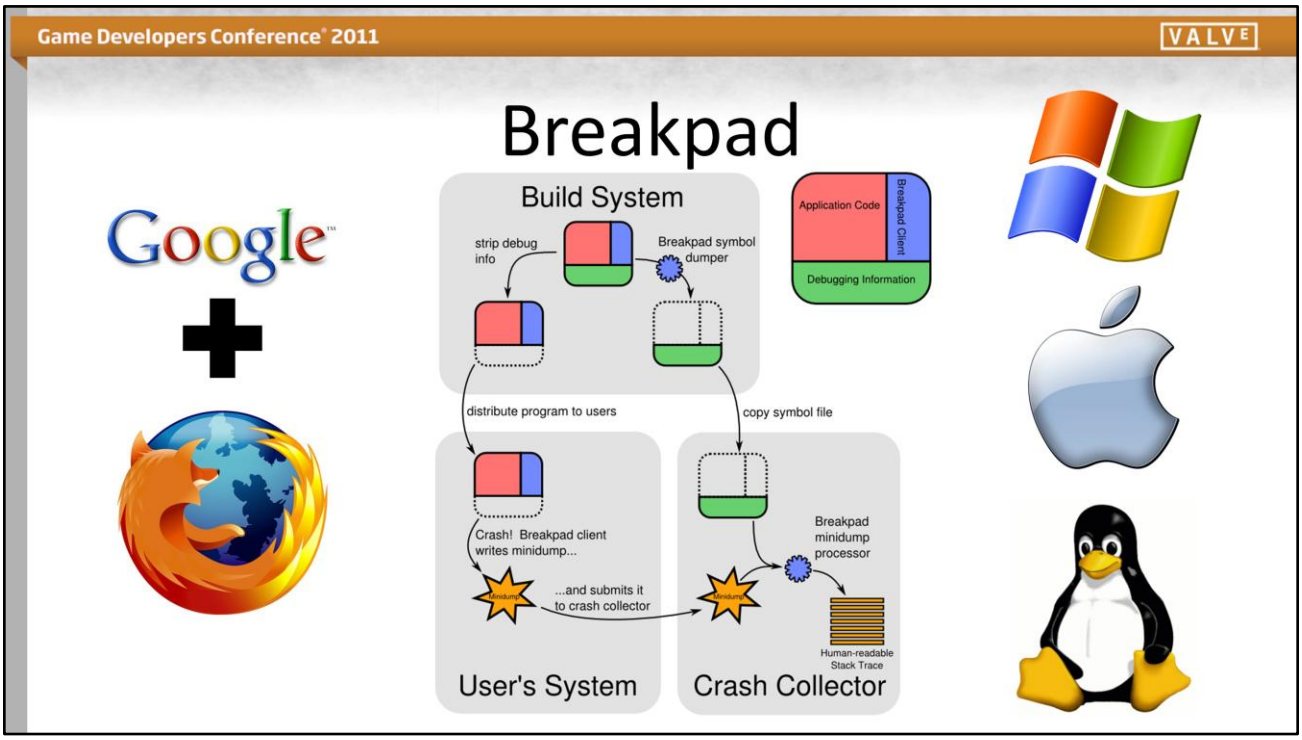
Built in ways:

Windows: Winqual. having the customer email you a .mdmp.

Mac: the Crash Report dialog, and asking customers to copy-and-paste the stack from it. That doesn't sound like a good time.



Replacing the default exception handler lets you write your game so that it does smart things with crashes, like emailing them to you.



How Valve uses Breakpad to send us data along the steam-tubes (with screenshots each step of the way)
Breakpad is a joint Google/Mozilla effort to create a multiplatform crash-reporting system. As used in Firefox.
We use it on all of our platforms, including our Linux dedicated game servers.

Game Developers Conference® 2011

VALVE

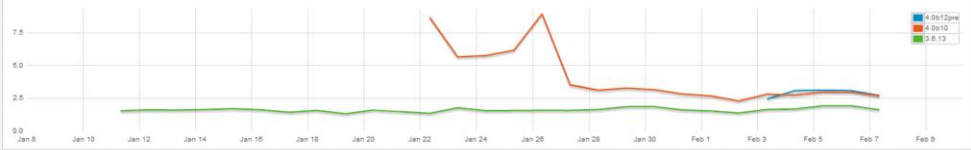
mozilla crash reports

Find Crash ID or Signature

Product: Firefox | Current Versions | Report: Overview | Advanced Search

Firefox Crash Data 3 days 7 days 14 days 28 days

Crashes per Active Daily User



Firefox Top Crashers | Top Crashers | Top Changers

Firefox 4.0b12pre	View all	Firefox 4.0b10	View all	Firefox 3.6.13	View all
(no signature)	198	mozcrt19.dll@0x1327f	44,103	UserCallWinProcCheckWow	144,927
extent_tree_ad_insert	175	mozalloc_abort(char const* const)	28,921	SkypeFfComponent.dll@0x440c3	113,429
extent_tree_ad_remove	150	js::mjit::EnterMethodJIT(JSContext*, JSStackFrame*, void*,	10,400	StrChrIA	62,464
mozalloc_abort(char const* const)	83	nsContentUtils::ASCIIToLower(nsAString_internal const&,	7,743	nsPluginInstanceOwner::PaintRect(tagRECT const&, HDC__*)	43,000
malloc_rtree_set	KERNELBASE.dll@0x659f	PL_DHashTableOperate	nsTHashtable<nsUniCharEntry	nsIFrame::GetOffsetTo(nsIFrame const*)	40,069
xpc::XrayWrapper<JSCrossCompartmentWrapper>::getJS	53	_PR_MD_SEND	5,281	SocketSend	29,717
nsGenericElement::GetScrollFrame(nsIFrame**)	49	(no signature)	4,943	(no signature)	29,598
zzz_AsmCodeRange_Begin	39	nsPluginInstanceOwner::CreateWidget()	4,528	nsGlobalWindow::cycleCollection::UnmarkPurple(nsISupp	28,953
huge_dallocx	33	FFPlugin.dll@0x19d0c4	4,181	JS_CallTracer	26,476
zero@0x5fddf	26	cairo_dwrite_font_face_create_for_dwrite_fontface	4,094	GraphWalker::DoWalk(nsDeque&)	26,434
zero@0x41667	26	GraphWalker<scan\Visitor>::DoWalk(nsDeque&)	3,982	NS_InvokeByIndex_P	24,856
js::mjit::EnterMethodJIT(JSContext*, JSStackFrame*, void*,	26	nsCSSFrameConstructor::SetUpDocElementContainingBl	3,397	js::TraceObject	24,672
zero@0x5fdd7	25	js::gc::MarkId	3,277	PR_AtomicIncrement	23,396
TextUpdater::FireEvent(nsAString_internal const&, unsigne	24	nsTextFrame::GetTrimmedOffsets(nsTextFragment const*	3,133	CCGraphBuilder::NoteXPCOMChild(nsISupports*)	23,043
malloc_rtree_set	chunk_alloc	mozalloc_handle_oom()	2,947	nsCycleCollectingAutoRefCnt::unmarkPurple()	21,493
js::ExecuteTree	24	StrChrIA	2,808	JS_TraceChildren	19,270
_PR_MD_SEND	22	PL_DHashTableOperate	2,756	icuuc36.dll@0x1f94	17,330
nsStandardURL::SetPath(nsACString_internal const&)	21	VirtualAllocEx	2,681	nsXULControllers::cycleCollection::UnmarkPurple(nsISupp	17,195
nsHTMLDocument::CreateShell(nsPresContext*, nsIViewM	20	js::Shape::trace(JSTracer*)	2,554	@0x0	SkypeFfComponent.dll@0x440c3	16,453
nsINode::doRemoveChildAt(unsigned int, int, nsIContent*,	19	js::gc::MarkObject	2,525	MWSBAR.DLL@0x474b7	15,403

This is Mozilla's public stability site. Anyone can see every crash that Firefox has suffered recently, its call stack, many details. Moz uses this to collect stability data from every Firefox in the world.

Mozilla blocks Skype add...

arstechnica.com/open-source/news/2011/01/mozilla-blocks-skype-add-on-caused-33k-firefox-crashes-in-a-week.ars?utm_source=rss&utm_

Remember The Milk... Bug entry template My Delicious Bookmark on Delici... Readability IEEE-754 Floating-P... cPanel® 11 Google Reader - Play

Mozilla blocks Skype add-on: caused 33k Firefox crashes in a week

By Ryan Paul | Last updated 17 days ago

Mozilla announced yesterday that it will block the Skype Toolbar add-on for Firefox and remotely disable it for existing users. Mozilla was forced to take this extraordinary measure after discovering that severe bugs in the add-on are crippling the browser's performance and stability.

The Skype Toolbar add-on is developed by Skype and comes bundled with the company's popular chat program. The add-on appears to be injected into Firefox automatically during the Skype installation and update process. Its primary function is to identify strings of text in Web pages that look like phone numbers and transparently convert them to links that can be used to automatically dial a call with Skype.

In November, a Mozilla engineer noticed half a dozen reports in the Firefox bug tracker that involved problems caused by the Skype Toolbar. A meta-bug was established to track the issues collectively and facilitate discussion about potential remedies. Mozilla's crash report system also identified the Skype Toolbar as one of the leading causes of Firefox crashes.

Some of the problems that Mozilla uncovered are fairly serious. The toolbar apparently performs its phone number conversion routine after every single DOM mutation, severely impairing the browser's performance. In some builds, the performance hit is so bad that it makes DOM manipulation 300 times slower. The add-on's misbehavior also reportedly causes rendering problems in a number of scenarios.

In particular, this helps Mozilla track down third party apps that suddenly cause problems for a disproportionate number of their users – because they have data on the actual call stack, and on all the modules loaded at the time, they can immediately pinpoint exactly what caused the crash and take action.

Breakpad Files

Dumps

- Microsoft .mdmp format
- Contain:
  - Exception cause
  - List of running threads
  - CPU registers and stack data
  - List of loaded .DLLs
  - Processor type, OS version
  - Comment field

Symbol files

- Breakpad's own (well documented) format
- Processed from PDB/.ELF/DWARF/STABS
- Build your own symbol server

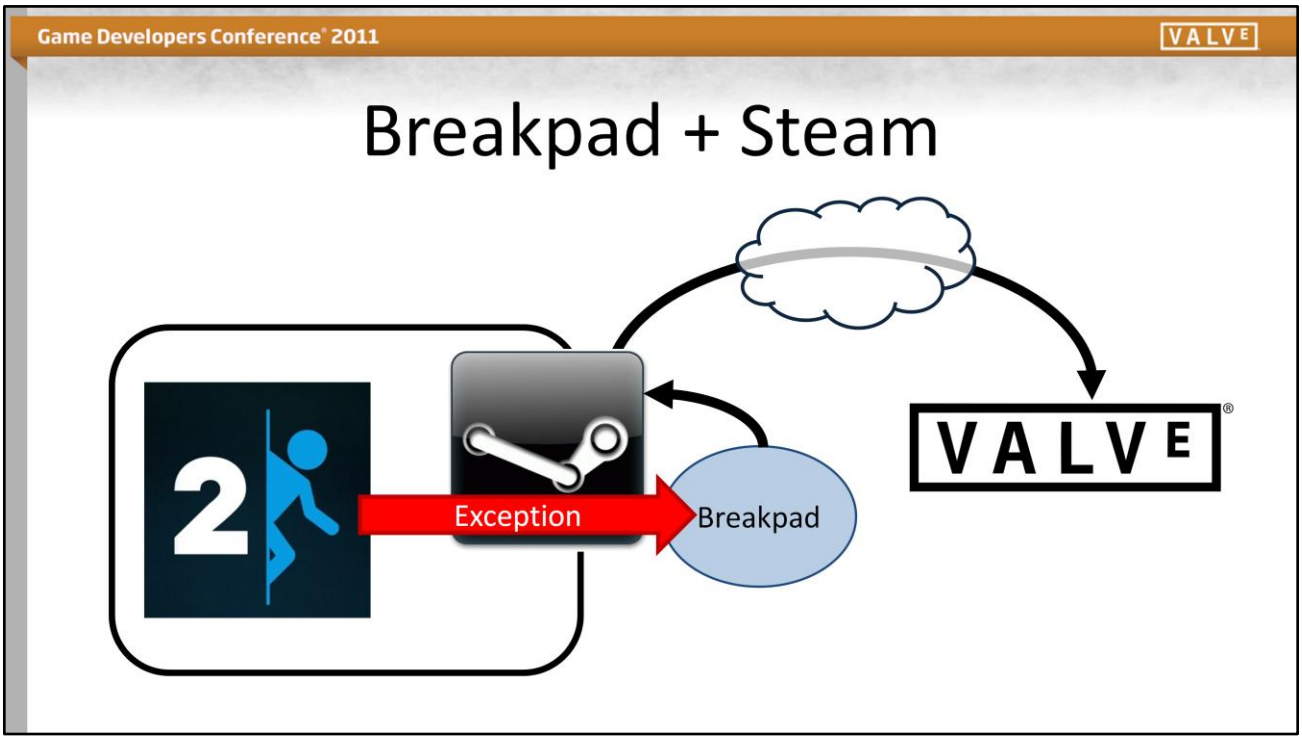
The minidump file format is similar to core files but was developed by Microsoft for its crash-uploading facility. A minidump file contains:
A list of the executable and shared libraries that were loaded in the process at the time the dump was created. This list includes both file names and identifiers for the particular versions of those files that were loaded.
A list of threads present in the process. For each thread, the minidump includes the state of the processor registers, and the contents of the threads' stack memory. These data are uninterpreted byte streams, as the Breakpad client generally has no debugging information available to produce function names or line numbers, or even identify stack frame boundaries.
Other information about the system on which the dump was collected: processor and operating system versions, the reason for the dump, and so on.

Breakpad Client Libraries

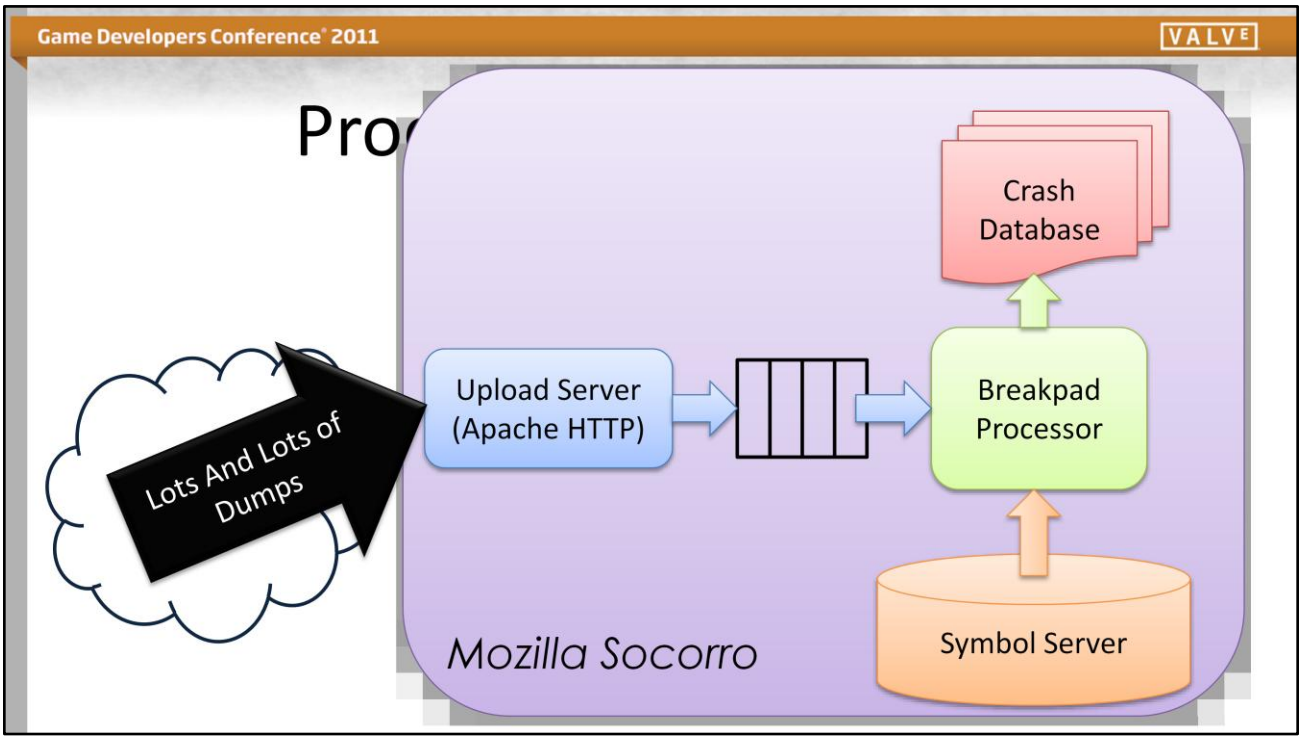
- Exception handler
  - (OS exception, not try/catch)
  - Installed at app launch
- Stackwalker
  - Runs in own thread (ideally process)
- Dump emitter
- HTTP uploader (optional)

Here is how we integrate the Breakpad client libraries into our games that we ship to customers.
The client library consists of an exception handler – an OS-level structured exception handler on Windows, not a try/catch block – and signal handler on Darwin and Linux.

http://code.google.com/p/google-breakpad/wiki/ClientDesign

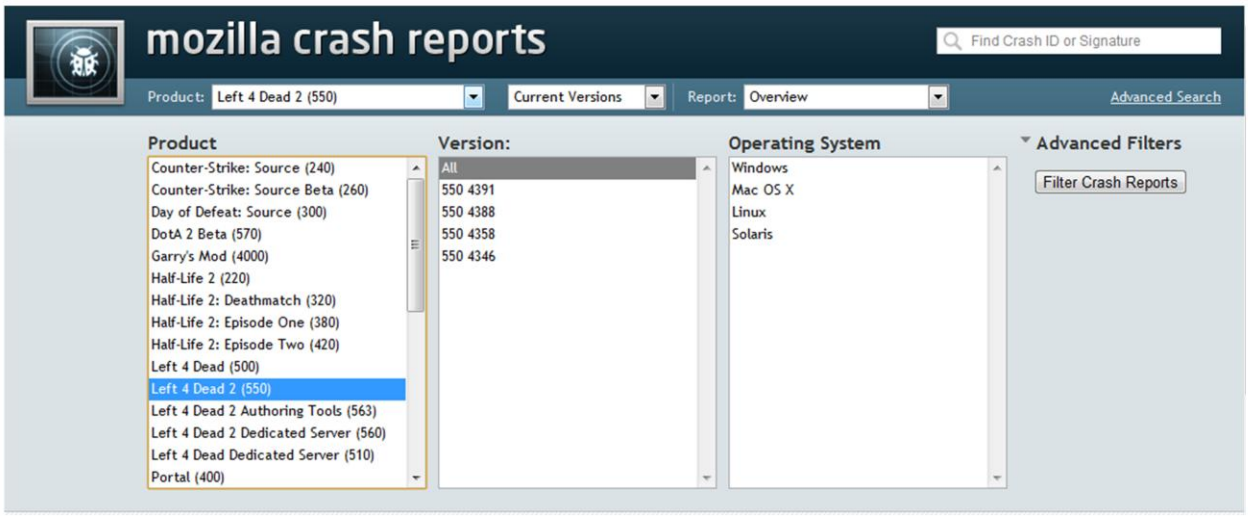


In the case of our games, since they run inside the Steam process, what actually happens is the game's exception percolates up to Steam, which has Breakpad installed. That exception then goes back to Steam, which uploads it to us!



The dumps all come in (at a very high rate) to a simple Apache server, which just accepts the uploads and queues them up. They are fed at a slower pace to our processor machine, which calls the Breakpad processor encapsulated as a MinidumpProcessor C++ class. Using the filename and timestamp of the modules the user was running, it finds the relevant symbol files on our server (the big flat filesystem again) and produces a call stack, which then gets uploaded to a crash database. This whole backend is actually another open source project from Mozilla called Socorro, which is what they use for their stability website.

CrashDB - Socorro



Storing hundreds of thousands of customer crash reports in a database. This is what our database looks like. As you can see, we're just using Mozilla's back end, called Socorro.

Query Results

Results within 1 weeks of 02/16/2011 02:05:03, and the product is one of 550 and the crashing process was a any.

Rank	Signature	#	Win	Mac	Lin	Sol
1	CAudioDirectSound::ClearBuffer()	6165	6165	0	0	0
2	(empty signature) Learn More	1962	0	1962	0	0
3	DSP_ClearState	1886	1886	0	0	0
4		1020	1020	0	0	0
5		752	752	0	0	0
6		718	718	0	0	0
7	CShaderAPIDx8::TexLock(int, int, int, int, int, int, CPixelWriter&)	459	459	0	0	0
8	Error | SV_InitGameDLL()	444	444	0	0	0
9		380	380	0	0	0
10	CShaderDeviceDx8::CreateStaticMesh(unsigned __int64, char const*, IMaterial*, VertexStreamSpec_t*)	312	312	0	0	0
11	CMatCallQueue::CallQueued()	281	281	0	0	0
12	Error | UTIL_SetModel(CBaseEntity*, char const*)	279	279	0	0	0
13		256	256	0	0	0
14		247	247	0	0	0
15	CMeshDX8::Lock(int, bool, VertexDesc_t&)	237	237	0	0	0
16	CMaterial::GetColorModulation(float*, float*, float*)	220	220	0	0	0
17	DecalSurfaceAdd(msurface2_t*, int)	217	217	0	0	0
18	CMeshDX8::SetPrimitiveType(MaterialPrimitiveType_t)	209	209	0	0	0
19	CParticleCollection::LabelTextureUsage()	168	168	0	0	0
20	Sys_Error | Host_ParseConfiguration	161	0	161	0	0
21	ZombieManager::StartFrame	147	0	0	147	0
22		141	141	0	0	0

Here's what you get if you click go; you can see here the list of crashes and their causes by count. Notice also that we have some crashes that occur only on Mac and another that's only on Linux.

Game Develope

Top Crashers for 550 4448

By Domain | By Signature | By Topsite | By URL

VALVE

Top 300 Crashing Signatures. 2011-01-18 20:00:00 through 2011-02-15 20:00:00. The report covers 93.90% of all 5381 crashes during this period. Graphs below are dual-axis, having Count (Number of Crashes) on the left X axis and Percent of total of Crashes on the right X axis.

Other Periods:

3 Days 7 Days 14 Days

Rank	Trend	%	Diff	Signature	Count	Win	Mac	Lin
1	new	33.40%	0.00%	CAudioDirectSound::ClearBuffer()	1797	1797	0	0
2	new	8.60%	0.00%	(empty signature) Learn More	463	0	463	0
3	new	2.88%	0.00%		155	155	0	0
4	new	2.86%	0.00%		154	154	0	0
5	new	2.27%	0.00%	DSP_ClearState	122	122	0	0
6	new	2.10%	0.00%	CShaderAPIDx8::TexLock(int, int, int, int, int, int, CPixelWriter&)	113	113	0	0
7	new	1.62%	0.00%		87	87	0	0
8	new	1.49%	0.00%	Error | SV_InitGameDLL()	80	80	0	0
9	new	1.36%	0.00%	CMatCallQueue::CallQueued()	73	73	0	0
10	new	1.34%	0.00%	CShaderDeviceDx8::CreateStaticMesh(unsigned __int64, char const*, IMaterial*, VertexStrea	72	72	0	0
11	new	1.30%	0.00%		70	70	0	0
12	new	1.15%	0.00%	Error | UTIL_SetModel(CBaseEntity*, char const*)	62	62	0	0

Game Developers Conference® 2011

VALVE

Windows NT - Crash Reports for CMaterial::GetColorModulation(float*, float*, float*)

Results within 2 weeks of 02/16/2011 02:54:35, and the product is one of 550 and the crashing process was a any.

Graph | Table | Reports | Comments (0) | Correlations

501 Crash Reports

Date	Product	Version	Build	OS	CPU	Reason	Address	Hang	Uptime	Comments
Feb 16, 2011 02:43	550	4448	20110111175313	Windows NT 5.1.2600 Service Pack 3	x86	EXCEPTION_ACCESS_VIOLATION_READ	0x10		0	
Feb 16, 2011 01:23	550	4448	20110111175313	Windows NT 6.1.7600	x86	EXCEPTION_ACCESS_VIOLATION_READ	0x10		0	
Feb 15, 2011 14:17	550	4448	20110111175313	Windows NT 6.1.7600	x86	EXCEPTION_ACCESS_VIOLATION_READ	0x10		0	
Feb 15, 2011 13:38	550	4448	20110111175313	Windows NT 6.1.7600	x86	EXCEPTION_ACCESS_VIOLATION_READ	0x10		0	
Feb 15, 2011 13:24	550	4448	20110111175313	Windows NT 6.1.7600	x86	EXCEPTION_ACCESS_VIOLATION_READ	0x10		0	
Feb 15, 2011 13:06	550	4448	20110111175313	Windows NT 6.1.7600	x86	EXCEPTION_ACCESS_VIOLATION_READ	0x10		0	
Feb 15, 2011 12:45	550	4448	20110111175313	Windows NT 6.1.7601 Service Pack 1, v.178	x86	EXCEPTION_ACCESS_VIOLATION_READ	0x10		0	
Feb 15, 2011 12:24	550	4448	20110111175313	Windows NT 5.1.2600 Service Pack 3	x86	EXCEPTION_ACCESS_VIOLATION_READ	0x10		0	

I can drill into a single crash cause, which is all the dumps that share the same call stack. Here you can see all those dumps.

Game Develope

VALVE

550 4448 Crash Report [@ CMaterial::GetColorModulation(float*, float*, float*)]

ID: 92ec2d6d
Signature: CMaterial::GetColorModulation(float*, float*, float*)

Search Mozilla Support for Help

Details | Modules | Raw Dump | Minidump Comment

Signature	CMaterial::GetColorModulation(float*, float*, float*)
UUID	92ec2d6d
Time	2011-02-15 13:38:41.885219
Uptime	0
Product	550
Version	4448
Build ID	20110111175313
Branch	steam
OS	Windows NT
OS Version	6.1.7600
CPU	x86
CPU Info	AuthenticAMD family 15 model 75 stepping 2
Crash Reason	EXCEPTION_ACCESS_VIOLATION_READ

Crashing Thread

Frame	Module	Signature [Expand]	Source
0	materialsystem.dll	CMaterial::GetColorModulation	//valvegames/rel/left4dead2/src/materialsystem/cmaterial.cpp:2068
1	shaderapidx9.dll	CBaseMeshDX8::DrawMesh	//valvegames/rel/left4dead2/src/materialsystem/shaderapidx9/meshdx8.cpp:2372
2	shaderapidx9.dll	CDynamicMeshDX8::DrawInternal	//valvegames/rel/left4dead2/src/materialsystem/shaderapidx9/meshdx8.cpp:4240
3	shaderapidx9.dll	CDynamicMeshDX8::Draw	//valvegames/rel/left4dead2/src/materialsystem/shaderapidx9/meshdx8.cpp:4253
4	vguimatsurface.dll	CMatSystemSurface::DrawQuad	//valvegames/rel/left4dead2/src/vguimatsurface/matsystemsurface.cpp:1109
5	vguimatsurface.dll	CMatSystemSurface::DrawTexturedRect	//valvegames/rel/left4dead2/src/vguimatsurface/matsystemsurface.cpp:1652
6	gameui.dll	gameui.dll@0x564fe	

Show/hide other threads

Drill into an individual dump and you can see data on the machine that suffered it, and the whole call stack.

Details | Modules | Raw Dump | Minidump Comment

Filename	Version	Debug Identifier	Debug Filename
launcher.dll		F842B628C67B404E9A448A8DC94202073	launcher.pdb
tier0.dll		7F9BD272AB8249A991C895508D58C18B1	tier0.pdb
filesystem_stdio.dll		F7CFC5A5F29E43819E9788FD2BE1B2863	filesystem_stdio.pdb
left4dead2.exe		C2DACBA6EF724787AFFF9DDBEB727BE81	default.pdb
vstdlib.dll		443E08D121F14C58B969A57F378831EA2	vstdlib.pdb
inputsystem.dll		E68D75B6E7064D4FA3D77C23335189F44	inputsystem.pdb
datacache.dll		34C950851BA646F5B1D4CF7FE2F986673	datacache.pdb
engine.dll		455D1F2174184D1FB17C263C9204A4E73	engine.pdb
materialsystem.dll		2E5AAF69064745ADA379473B108528DF3	materialsystem.pdb
studiorender.dll		94E8C64AF7E24FBF9181081652A657313	studiorender.pdb
vphysics.dll		2C3C1E15B4AE49239CCBE6FADEE2C4AF3	vphysics.pdb
vscript.dll		723763006A594ED09C63EA02B0EDF8883	vscript.pdb
valve_avi.dll		983E989E5F114996921E60EF5C953DC43	valve_avi.pdb
XInput1_3.dll	9.15.779.0	3482C2EF26164FFC863CC4E4BBA3210A1	XInput1_3.pdb
vguimatsurface.dll		1D278BADB24842469B3DB9918D29BDF03	vguimatsurface.pdb
vgui2.dll		49D1D62458344D40B4150FA4C92DF4EC3	vgui2.pdb
crashhandler.dll	1.1.0.43	6E7B1B193D5843CB8B48D9D1CDE1D7AD1	crashhandler.pdb
shaderapidx9.dll		F8848616C14940559282FCDC76073B323	shaderapidx9.pdb
stdshader_dbg.dll		6BFEA17519784E80A27DD76B7B67D70B3	stdshader_dbg.pdb
unicode.dll		8CE01DA3B5E14340841C651AA1977DB03	unicode.pdb
stdshader_dx9.dll		CC80C7E5CC5647A59E55635DBF8C31303	stdshader_dx9.pdb
clbcatq.dll	2001.12.8530.16385	00A720C79BAC402295B6EBDC147257182	CLBCatQ.pdb
msssrs.flt	7.2.6.0	97F4F28931D94708BDD5ED1C2D51D3751	msssrs.pdb
vaudio_miles.dll		3528962A7424452F94553269012213C63	vaudio_miles.pdb
msseax.flt		4E23FE81FDA14EC3951D43C77EDBF9CC1	msseax.pdb

You can see all the DLLs that were loaded and their version.

Details | Modules | Raw Dump | Minidump Comment

```
waterforceexpensive: 0
waterforcereflectentities: 0
mat_motion_blur_enabled: 1
mat_queue_mode -1
mat_triplebuffered: 0


Memory
memusage( 44 % )
totalPhysical Mb(3071.43)
freePhysical Mb(1704.09)
totalPaging Mb(6141.01)
freePaging Mb(4501.71)
totalVirtualMem Mb(4095.88)
freeVirtualMem Mb(3140.30)
extendedVirtualFree Mb(0.00)

Paged Pool
prev PP PAGES: used: 0, free 0
final PP PAGES: used: 0, free 0
memallocfail? = 0
Active: active SpawnCount 2 MapLoad Count 2
Error count 0, end demo 0, abort count 0
```

And, if you remember that I mentioned earlier that you can add any blob of “comment” data that you like to a minidump, we use ours to store useful telemetry, like the OS configuration the user had, how much free memory, and so on.

Game Develope

VALVE

550 4448 Crash Report [@ CMaterial::GetColorModulation(float*, float*, float*)]

ID: 92ec2d6d
Signature: CMaterial::GetColorModulation(float*, float*, float*)

Search Mozilla Support for Help

Details | Modules | Raw Dump | Minidump Comment

Signature	CMaterial::GetColorModulation(float*, float*, float*)
UUID	92ec2d6d
Time	2011-02-15 13:38:41.885219
Uptime	0
Product	550
Version	4448
Build ID	20110111175313
Branch	steam
OS	Windows NT
OS Version	6.1.7600
CPU	x86
CPU Info	AuthenticAMD family 15 model 75 stepping 2
Crash Reason	EXCEPTION_ACCESS_VIOLATION_READ

Crashing Thread

Frame	Module	Signature [Expand]	Source
0	materialsystem.dll	CMaterial::GetColorModulation	//valvegames/rel/left4dead2/src/materialsystem/cmaterial.cpp:2068
1	shaderapidx9.dll	CBaseMeshDX8::DrawMesh	//valvegames/rel/left4dead2/src/materialsystem/shaderapidx9/meshdx8.cpp:2372
2	shaderapidx9.dll	CDynamicMeshDX8::DrawInternal	//valvegames/rel/left4dead2/src/materialsystem/shaderapidx9/meshdx8.cpp:4240
3	shaderapidx9.dll	CDynamicMeshDX8::Draw	//valvegames/rel/left4dead2/src/materialsystem/shaderapidx9/meshdx8.cpp:4253
4	vguimatsurface.dll	CMatSystemSurface::DrawQuad	//valvegames/rel/left4dead2/src/vguimatsurface/matsystemsurface.cpp:1109
5	vguimatsurface.dll	CMatSystemSurface::DrawTexturedRect	//valvegames/rel/left4dead2/src/vguimatsurface/matsystemsurface.cpp:1652
6	gameui.dll	gameui.dll@0x564fe	

Show/hide other threads

So let’s see what caused this particular issue.

```
2063.   void CMaterial::GetColorModulation( float *r, float *g, float *b )
2064.   {
2065.           PrecacheVars();
2066.
2067.           float pColor[3];
2068.           m_pShaderParams[COLOR]->GetVecValue( pColor, 3 );
2069.           *r = pColor[0];
2070.           *g = pColor[1];
2071.           *b = pColor[2];
2072.   }
2073.
```

In this case it looks like the issue is due to missing shader parameters. So we ought to build a machine with that same configuration of graphics card and video driver and see why this shader isn't initializing properly.

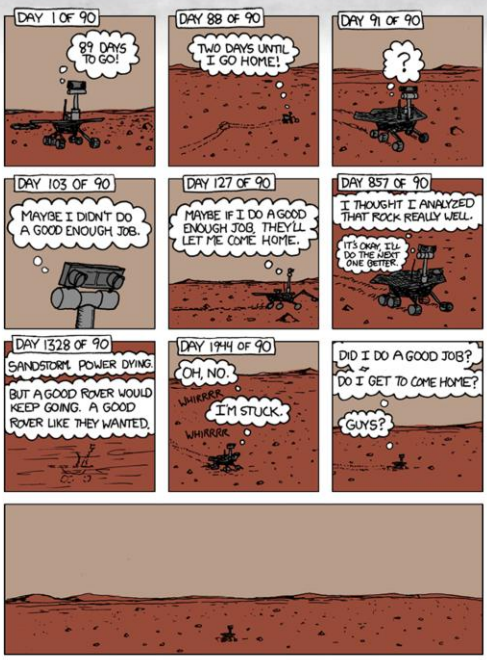
In Conclusion

- Crashes suck.
- Fixing them doesn't have to.
- When everyone else lies, the disassembler is there for you.
- These skills are for debugging live release builds too.
- "The debugger should" ≠ "The debugger **does**".

GO FORTH AND CRASH NO MORE!

slides at:
bit.ly/hPCmVW

Questions?



XKCD by Randall Munroe

Bibliography / Cheat Sheet

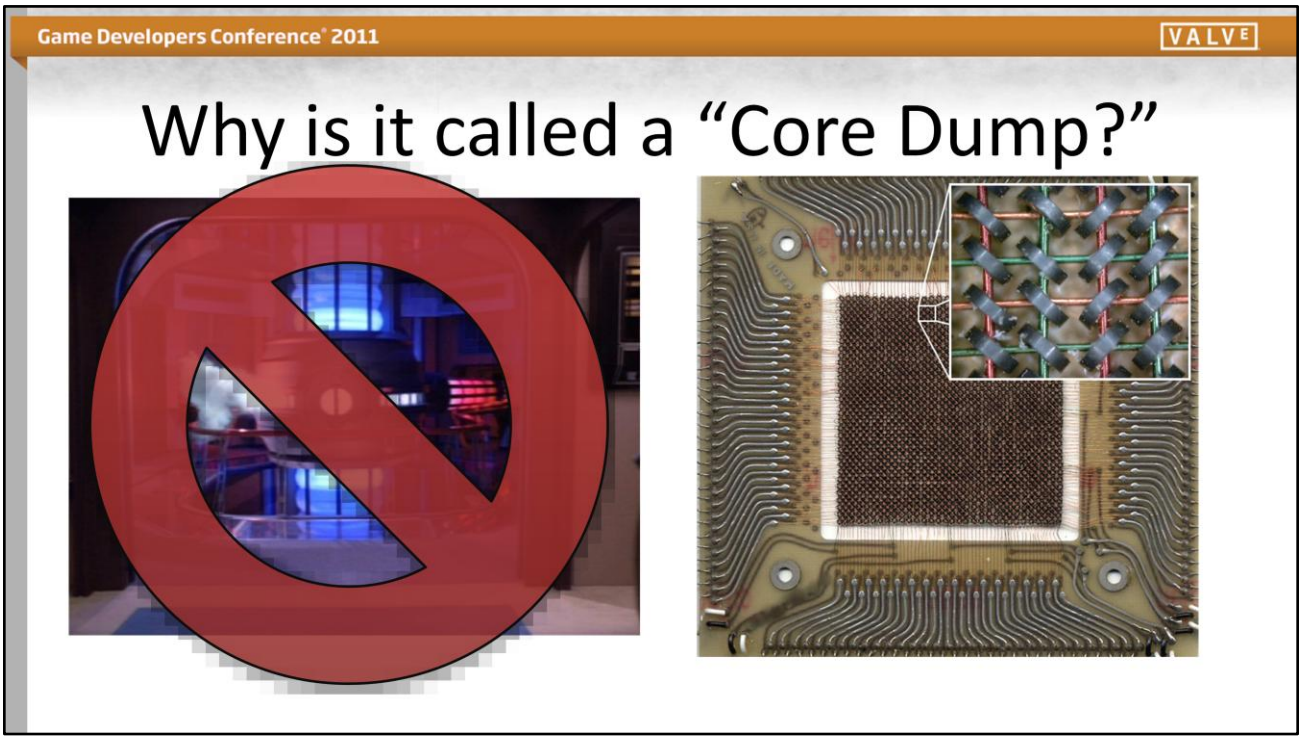
- Pietrek, Matt. *Just Enough Assembly Language To Get By, I & II.* Microsoft Systems Journal, Feb-June 1998.
  bit.ly/9HKCOk bit.ly/bJf1R0
- (Microsoft) Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. *Debugging in the (Very) Large: Ten Years of Implementation and Experience*.
  bit.ly/4lBgkH
- (Jet Propulsion Laboratory) Reeves, Glenn; Neilson, Tracy. *The Mars Rover Spirit FLASH Anomaly*.
  http://hdl.handle.net/2014/39361
- My blog: *Some Assembly Required*.
  http://assemblyrequired.crashworks.org
  slides for this talk: bit.ly/hPCmVW

Bibliography / Cheat Sheet

- IBM. *PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors.* http://bit.ly/gHemWI
- Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual.* http://www.intel.com/products/processor/manuals/
- SunSoft; IBM. *SYSTEM V APPLICATION BINARY INTERFACE PowerPC Processor Supplement.* http://bit.ly/fRvDe9
- Apple. *Introduction to Mac OS X ABI Function Call Guide.* http://bit.ly/foRCnE
- Motorola, Inc. *PowerPC Embedded Application Binary Interface.* http://bit.ly/i9GOyz
- IBM. *Developing PowerPC Embedded Application Binary Interface (EABI) Compliant Programs.* http://bit.ly/hbhlyk
- Taylor, Ian Lance (Zembu Labs). *64-bit PowerPC ELF Application Binary Interface Supplement.* http://bit.ly/hoWPfF
- *ELF-64 Object File Format.* http://bit.ly/fUel3t
- Wikipedia. *Executable and Linkable Format.* http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Bibliography / Cheat Sheet

- Eager, Michael. *Introduction to the DWARF Debugging Format.* http://bit.ly/hLBoF2
- Apple. *Debugging and Symbolizing Crash Dumps in Xcode.* http://bit.ly/C2RqC
- Microsoft. *Symbol Server and Symbol Stores.* http://bit.ly/ecSyzj
- Microsoft. *Debugging Tools for Windows.* http://bit.ly/eXwcf2
- WinDbg command cheat sheet: http://bit.ly/3Jc0p7
- Sen, Saikat. *A WinDbg Tutorial.* http://bit.ly/4h4PZx
- Google Breakpad project home page: http://bit.ly/7FfWzQ
- http://www.dumpanalysis.org/



By the way, did you ever wonder why it is called a core dump? I thought it was because the people who invented Unix were big nerds and made some kind of Star Trek reference, but not so. In fact it’s because before computers used capacitors for their RAM, they used ferrite cores – little donuts of iron – to store bits magnetically. So, memory was literally called core, and if the computer crashed, it would dump the entire contents of core memory to disk, or the punchcard writer. Hence, core dump.

Whirlwind I

And if you think that's bad, when the 1951 Whirlwind I computer crashed, it would output the entire core memory to a CRT, in octal. Then an automated camera would take a picture of the CRT on microfilm, which would be developed and sent over to the poor sap debugging it.

Socorro

- Mozilla's back-end for stability data
- Socorro Server
  - Collects/databases/processes dumps
- Socorro UI
  - The UI you just saw
- Python, Postgres, Hadoop, …
- You could write your own DB/UI too.

http://code.google.com/p/socorro/

A web front-end to your crash database: Socorro.
Another open-source tool from Mozilla.
There are pros and cons to using it; you could also write your own front end.

we've made minor modifications to the socorro code - it's PHP + Postgres. We've added things like steam universe so we can filter on it. we're *NOT* using the hadoop crap - we tried, and gave up - their current trunk version is overly complex to setup and administer. we backed off to the last "stable" version where they used NFS for dump transmission and didn't rely on hadoop, and the system has been maintenance free (knock on wood) since.

http://code.google.com/p/socorro/wiki/SocorroOverview

x86 frame pointer omission

address		
0x10000	caller EBP	ebp
0xfffc	int a	esp+42
0xfff8	int b	esp+38
0xfff4	float f	esp+34
0xfff0	float g	esp+30
0xffec		
0xffe8	uint64 u	esp+24
0xffe4		
0xffe0		
0xffdc		
0xffd8		
0xffd4		
0xffd0	m128 vec	esp+0

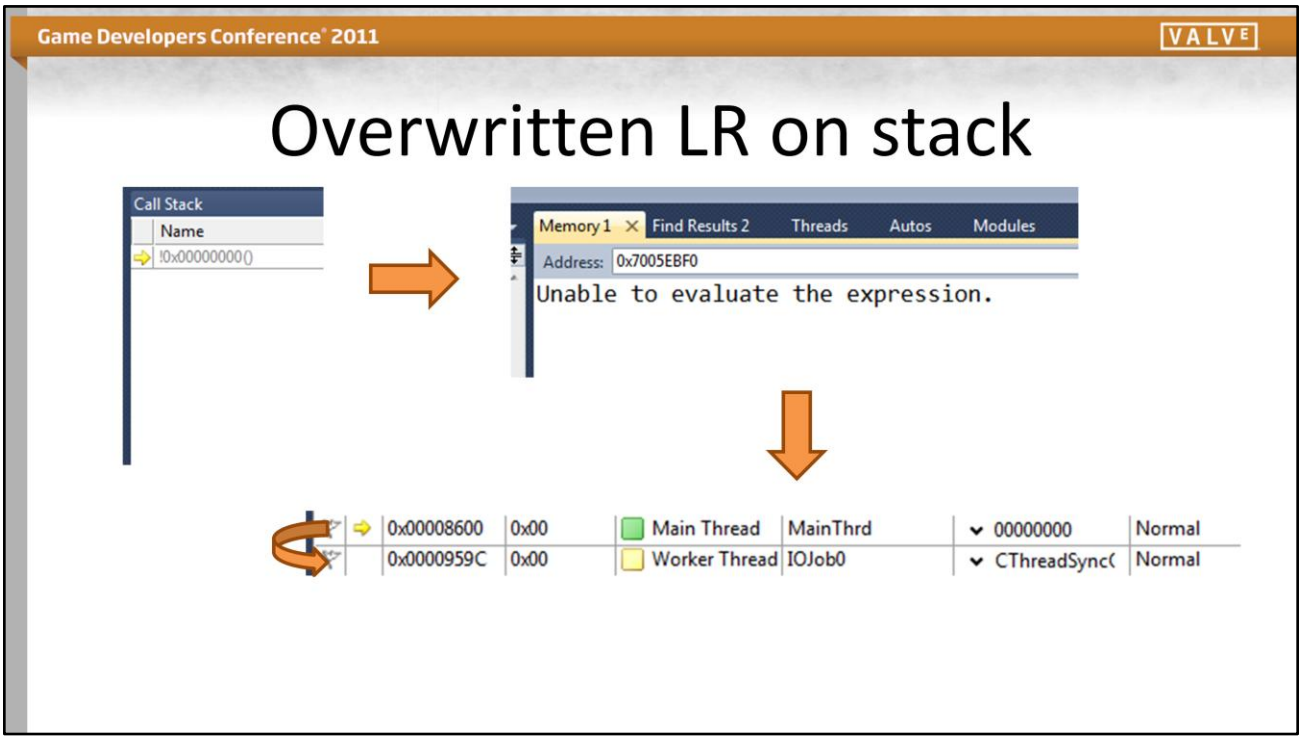
EBP

ESP

```
void foo()
{
    int a,b;
    float f,g;
    uint64 u;
    __m128 vec;
}
```

```
mov eax, [ebp+42] ; a
add eax, [ebp+38] ; a+b

fld [ebp+34] ; f
fld [ebp+30] ; g
fsub         ; f - g

movaps xmm0, [ebp+0]; ; sse vec load
```

No nonvolatile registers on x86, but debugger is good about moving the stack pointer around properly. If you see that the EBP pointer here is redundant, you're right: enabling "Frame Pointer Omission" in the compiler does away with it, using just the ESP pointer instead. The trouble with that is you lose your backchain – the compiler has to know how much to move the stack pointer when returning from each function, and the debugger doesn't always have that info – and it complicates finding data since the ESP pointer tends to move about spastically as a consequence of push/pop.



A quirk of MSVC is that when your stack frame thinks it's at address 0x00, it may fail to give you a memory window altogether. The solution is simply to switch to a different thread that still has a valid context, and then memory will come back.

Breakpad Exception Handling

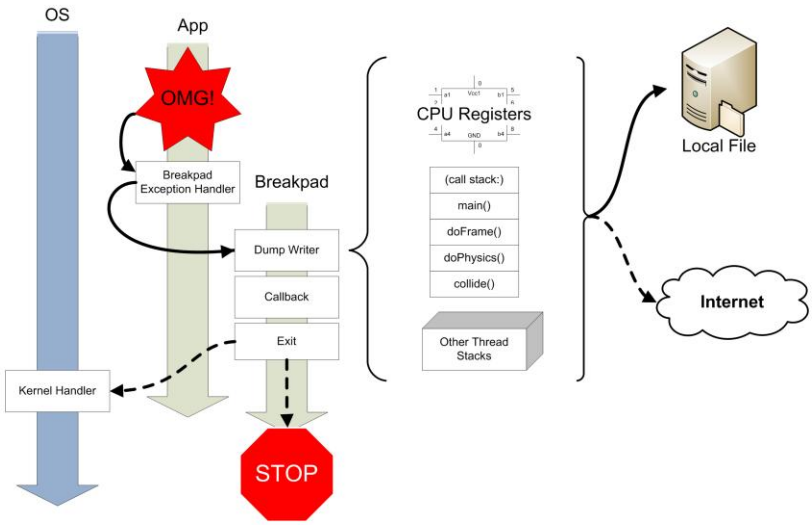


Photo Credits

NIH / National Library Of Medicine, Visible Proofs exhibit
Wikimedia Commons public domain archive
Library Of Congress Photo Archive
United States Air Force
NASA

Chalk outline: Flickr user rbeiber
http://www.flickr.com/photos/rbieber/155102957/
Lying rat: Flickr user niznoz
http://www.flickr.com/photos/niznoz/4233333
Memphis sewer: Flickr user mojorider2 / Paul Everett
http://www.flickr.com/photos/mojorider2/4670627480/

3c32566u.tif
Title: United States Volunteers attacked by the mob, corner of Fifth and Walnut Streets, St. Louis, Missouri / sketched by M. Hastings, Esq.
Date Created/Published: 1861.
Medium: 1 print : wood engraving.
Summary: Soldiers firing at crowd of people attacking them.
Reproduction Number: LC-USZ62-132566 (b&w film copy neg.)
Rights Advisory: No known restrictions on publication.
Call Number: Illus. in AP2.H32 1861 (Case Y) [P&P]
Repository: Library of Congress Prints and Photographs Division Washington, D.C. 20540 USA
Notes:
Illus. in: Harper's weekly, v. 5, no. 231 (1861 June 1), p. 349 (botttom).

F2A_Thach_Accident.jpg
Description
English: On 19 March 1940 U.S. Navy Lt. John Smith "Jimmy" Thach tipped this Brewster F2A-1 Buffalo (BuNo 1393) onto its nose on the flight deck of the circraft carrier USS Saratoga (CV-3) Ensign Edward Butch O'Hare also flew this aircraft several times during the summer and fall of 1940.
Date
19 March 1940
Source
broadcast.illuminatedtech.comTransferred from en.wikipedia
Author
USN; Original uploader was Felix c at en.wikipedia, 25 June 2006 (original upload date)
Permission
(Reusing this file)
PD-USGOV-MILITARY-NAVY.

chalk_outline.jpg
Flickr cc-nc
user rbeiber
http://www.flickr.com/photos/rbieber/155102957/

nih_nyc_crime_scene.jpg
NIH Web Site
New York City crime scene, 1914-1918
New York City Municipal Archives
http://www.nlm.nih.gov/visibleproofs/exhibition/views.html

af_c141_ramp_crash
US Air Force
http://www.af.mil/photos/media_search.asp?q=crash&page=14

Ferrite_core_memory
Wikipedia public domain
Description
English: Random access ferrite core memory (RAM) from 1961. Size of the card : 10.8cm x 10.8cm (6.5 inch), capacity : 1024 bits.
Date: 5 June 2009
Source: Combined from Magnetic core memory card.jpg and Magnetic core.jpg.
Author: Orion 8

preliminary_incision
Upon A View of the Body
NIH Visible Proofs exhibition
"Fig. 5: Reflexion of Thoracic Coverings: Take up a position on the right side of the body. The head of the corpse should be extended and the chin steadied with the left hand. Grasping the post-mortem knife firmly in the palm of the right hand and cutting with the belly, not the point, of the knife, make a median incision from the chin to the pubes."
Charles Richard Box, M.D., Post-mortem Manual: A Handbook of Morbid Anatomy and Post-mortem Technique, London
National Library of Medicine
http://www.nlm.nih.gov/visibleproofs/galleries/exhibition/body.html

lying_rat
flickr user niznoz
http://www.flickr.com/photos/niznoz/4233333

Euklid
Justus of Ghent, 1474

whirlwind_control_room
Museum Of Science, Boston, MA
credit: Wikipedia

8867-Project-Whirlwind
credit: Wikipedia

KL_microsoft_xbox_360
credit: Wikipedia

nih_forensic_anthropology
Ethiopia, 1990s.
Doctors and personnel at the Black Lion Hospital in Addis Ababa work in the laboratory with members of the argentine Forensic Anthropology Team as part of their training in forensic anthropology.
National Library of Medicine
Credit: Stephen Ferry
http://www.nlm.nih.gov/news/press_releases/visibleproofphotos.html

nih_forensic_vials
Vials and Evidence, July 2004.
Dr. Andresen tested the soil around the caskets, and every type of embalming fluid used on the victims' bodies, to make sure there was no crosscontamination. Six exhumed patients tested positive for Pavulon. The proof was definitive—homicide had taken place. Saldivar had poisoned his victims.
National Library of Medicine
Credit: Courtesy of Anthony Pidgeon
http://www.nlm.nih.gov/news/press_releases/visibleproofphotos.html

nih_forensic_guy
Brian Andresen with a sample vial, July 2004. Dr. Andresen had to find some way to detect minute concentrations of Pavulon in long-buried victims—a method of teasing the drug out of decomposed tissue.
National Library of Medicine
Credit: Courtesy of Anthony Pidgeon
http://www.nlm.nih.gov/news/press_releases/visibleproofphotos.html

nih_fingerprint_diagram_iii_c_204
Fingerprint diagram, 1940
Frederick Kuhne, The Finger Print Instructor...Based upon the Sir E. R. Henry System of Classifying and Filing..., New York
National Library of Medicine
http://www.nlm.nih.gov/visibleproofs/galleries/exhibition/views_image_11.html

memphis_sewer.jpg
Creative Commmons
Flickr user mojorider2 / Paul Everett
http://www.flickr.com/photos/mojorider2/4670627480/

mars_spirit_rover_artists_rendition.jpg
NASA