the Availability Digest

www.availabilitydigest.com
@availabilitydig

160516

June 2016

To comply with full disclosure, we must point out that this article was submitted by an Availability Digest subscriber who wishes to remain anonymous.

160516. What is that?

It is a NonStop date routine bug that has lain dormant for over thirty years. Just after midnight of Friday, May 13, 2016, support personnel from several vendors were alerted by a NonStop vendor to a critical bug that would impact payment-processing applications on Monday, May 16th.

Several vendor products exhibited the very same date bug. When the date changed to May 16, 2016, the application processes would abend. They would continue to abend with an arithmetic overflow every time they were restarted.

What possibly could be special about May 16, 2016, so special that nobody saw it coming?

The line of code that caused the issue was:

```
int day := $UDBL (JULIAN^DAY^NUMBER '\' %177776 + 1) '\' 7 + 1;
```

This code calculates a day of the week (1-7) from a Julian day number. The Julian day is the number of days since January 1, 4714 B.C. The Julian day for Monday, May 15, 2016, is 2457525. This Julian day is what causes the overflow. One would assume that every date afterwards would cause an issue also, but none of them do. Date tests have been run out for weeks. Very large Julian days were tried. All functioned correctly.

This date routine is probably over 30 years old. It was written well before there was a Guardian call to do the same thing. Today a Guardian procedure exists to calculate the day of the week from a Julian day:

```
int day := DAYOFWEEK(julian^day^number);
```

Fortunately, most systems would not switch to a processing date of 160516 until the coming Sunday evening. The problem was identified by a payment-processing vendor that switched to the Monday processing date following completion of Fridays' processing in order to test its applications. The vendor's actual production impact was minimal, since it would not require OLTP capability until opening time Monday. The vendor immediately notified two other payment-processing vendors of the bug.

Application code was changed overnight on Friday with vendors pulling all-nighters to ensure code was built, tested, and ready for shipping to customers on Saturday. Developers and testers pulled out the stops that weekend to ensure customers would not be impacted by the impending issue. This was not a

risk of system failure - it was a certainty. A high number of ATM's would have been without service had the correction not been made in a timely manner.

The interesting question is what is wrong with that line of code?

The code does the following to find the day of the week:

> Divide Julian day by 7 then add 1.

Unfortunately, even in the 70s, the Julian day number was so big that the result of dividing it by 7 produces a quotient too large to be held in an integer. For instance,

> January 1, 1970 = the Julian day 2440588
> 2440588 / 7 = 248,655

What this code does is to make the number smaller. To do this, it divides the Julian day by the biggest number you can store in an unsigned integer (16 bits) that is exactly divisible by seven (ie %177776 = 65,534) and then adds 1, yielding a result of 65,535. In effect, it is dividing the Julian day by seven 9,362 times and then adding 1. The remainder will never be more than 65,533, which is just less than the maximum capacity of an unsigned integer (65,536). Thus, you can perform the divide by 7 and add 1 with a manageable number.

Unfortunately, the code in question used a signed add, and it is this bug that lay dormant for so long.

```
int day;
day := $UDBL (JULIAN^DAY^NUMBER '\' %177776 + 1) '\' 7 + 1;
```

The code above results in a arithmetic overflow error when applied to May 16, 2016, since the capacity of the unsigned integer is just 15 bits, or 32,768. The code below shows the error corrected. It uses an unsigned add operation:

```
int day;
day := $UDBL (JULIAN^DAY^NUMBER '\' %177776 '+' 1) '\' 7 + 1;
```

The faulty code has been found in applications produced by at least two different vendors. These applications in some form or another date back to when no Guardian procedure was available to calculate the day of week. Errors in other minor NonStop utilities were also observed on this date, indicating that the use of this code could be even more widespread.

Is it possible that the above method was published in an ITUG article or in a programming guide from HP? Is it possible that the same developer work for all three vendors?

Fortunately, testing indicates that the above routine, even if unfixed, will be behave correctly for at least the next 100 years.

Lesson Learned

If you have old applications that calculate the Julian day, replace that code with the Guardian procedure.

Acknowledgement

We would like to thank our anonymous subscriber for providing us the information contained in this article.