Steffen Müller

Das gefürchtete „Attempt to free unreference scalar"

Jedes mal wenn ich einen Fehler wie „Attempt to free unreferenced scalar: SV 0xDEADBEEF" sehe, rutscht mir das Herz in die Hose. Ich weiß, dass ich mit einer ausschweifenden Debugging-Sitzung dran bin. Das Debuggen von Speichermanagementproblem ist immer mühevoll. Das ist in Perl so und in jeder anderen Sprache. *valgrind* und ähnliche Tools können ein Geschenk des Himmels sein, aber auch sie kommen mit einigen Arten von Problemen nicht klar.

Das Problem verstehen

In den Perl-Diagnosemeldungen ist der folgende hilfreiche Absatz über die Warnung zu finden:

Perl hat versucht den Referenzzähler eines Skalars zu dekrementieren um zu prüfen ob er auf 0 gehen würde und hat dabei herausgefunden dass der Referenzzähler bereits 0 ist. Der Skalar hätte schon freigegeben werden sollen und wahrscheinlich wurde er schon freigegeben. Das könnte darauf hindeuten dass SvREFCNT_dec() zu häufig aufgerufen wurde oder das SvREFCNT_inc() zu selten aufgerufen wurde. Oder, dass das SV zu einem Zeitpunkt „mortalized" wurde als es noch nicht hätte passieren dürfen. Oder, dass der Speicher korrupt ist.

Lasst es uns etwas auseinander nehmen, da es einige Implementierungsdetails von Perl anspricht: Die aktuelle Implementierung von Perl5 verwendet ein referenzzählerbasiertes Speicherverwaltungsschema. Jeder Basiswert (ein Skalar, technisch gesehen ein Pointer auf ein SV-struct) hat einen Slot, der mitzählt, wie oft dieser Wert von etwas anderem referenziert wird. Wenn du eine neue Referenz auf einen SV erzeugst, erhöhst du diesen sogenannten *refcount*. Wenn du die Referenz auf den SV freigibst, musst du diesen Zähler verringern. Sobald dieser *refcount* 0 erreicht, gibt Perl den Speicher, der mit dem SV verbunden ist, frei und verringert dabei den refcount aller anderen SVs auf die der freigegebene SV eine Referenz hält. `SvREFCNT_inc()` und `SvREFCNT_dec()` sind die Perl-(C)-Makros, die genau das machen.

Wenn Du `SvREFCNT_inc()` einmal zu oft aufrufst oder `SvREFCNT_dec()` einmal zu wenig, dann „leaken" der SV und alles worauf dieser SV eine Referenz hält, weil sie nie zerstört werden - bis zur globalen Zerstörungsphase der `perl`-VM. Wenn Du das Gegenteil machst (zu viele `SvREFCNT_dec()`- oder zu wenige `SvREFCNT_inc()`-Aufrufe), wird der refcount des SV zu früh auf 0 gesetzt und es wird freigegeben, obwohl es immer noch von anderen Datenstrukturen referenziert wird. Leider sind das genau die, die in glückseliger Unwissenheit des bevorstehenden Fehlers durch einen ungültigen Speicherzugriff übrigbleiben.

Die oben erwähnte Warnung wird von Perl ausgegeben, wenn der Referenzzähler eines SV heruntergezählt wird und der Zähler bereits 0 ist. Beginnend mit dem Referenzzähler bei 0 bedeutet das, dass es nicht länger ein gültiger SV ist, der von Perl benutzt wird. Da das Speichersegment, in dem der SV gespeichert war, nicht länger zur Speicherung des originalen SV benutzt wird (siehe weiter unten für mehr Infos), könnte der Speicher mittlerweile für einen anderen SV benutzt worden sein. Sollte das so sein, wirst du die Warnung über den Skalar mit der schlechten Speicherverwaltung nicht sehen. Perl kennt Deine Absichten nicht und zählt fröhlich den Referenzzähler des neuen Speicherbewohners herunter. Das bedeutet schließlich, dass der Referenzzähler des neuen SV zu früh auf 0 gesetzt wird. Das wiederholt sich bis du es schaffst, den Speicher zu korrumpieren und davor zu warnen, bevor Perl eine Chance hat den Speicher wiederzuverwenden. Viel Spaß!

Aufmerksame C-Programmierer werden nun das Speicher-Debugging-Tool des Tages auf den Tisch bringen (mein Favorit ist und bleibt *valgrind/memcheck*), das mit dieser Art von Problemen durch Identifikation von ungültigen Zugriffen auf freigegebenen Speicher sehr effektiv umgeht. Ich wünschte es wäre so einfach! Das Schema ist doppelt falsch: Zum einen hat bei der oben gezeigten *Action at a distance* perfekt valider Code den ungültigen Speicherzugriff. Aber noch wichtiger ist: Das macht es Perls interner Speicherwaltung sehr wahrscheinlich, dass das Problem auftritt. Perl benutzt *slab allocation* um zu verhindern, wegen jedem einzelnen SV, das es erzeugt, über das Betriebssystem zu gehen, da viele *malloc*-Implementierungen der Betriebssysteme mangelhaft sind. Die Teile der SV-Struktur, die den Referenzzähler halten sind in so einem *slab* (in den Perl-Quellen als *arena* bezeichnet) zugewiesen, das typischerweise der Größe einer Speicherseite entspricht und so viele Elemente hält, wie in diese Größe passen. Perl verwendet eine Liste von unbenutzten Elementen in dem *slab* um effektiv SVs „zuweisen" und „freigeben" zu können. Das ist gut für die Performanz und um eine Fragmentierung des Speichers zu verhindern. Aber für das Debuggen von solchen Speicherproblemen verstärkt das die *Action at a Distance*-Problematik durch das häufige Wiederverwenden der SV *slab*s.

Wenn du Opfer der Warnung wirst, bringt eine Suche im Internet eine ganze Anzahl von Fällen zu Tage, in denen verzweifelte Leidensgenossen nach Hilfe von Experten fragen. Leider gibt es nicht das eine wahre Rezept zum Debuggen, das zu einer Lösung in allen möglichen Fällen führt und die wenigen spezifischen Hinweise, die es gibt, erfordern in der Regel, dass man eine spezielle Kopie von Perl für das Debugging baut.

Dein Perl aufrüsten

Es gibt eine Reihe von Optionen für `Configure`, die unterschiedlich gut in der Perl-Dokumentation abgehandelt werden, die dabei helfen eine Kopie von Perl zu bauen, das einige der oben genannten *Action at a distance*-Probleme vermeidet. Das Basis-Rezept (für *nix) zum Bauen deines eigenen Perls ist wie folgt (angenommen, du bist in einem Checkout des Perl-git-Repositories oder einem entpackten Release-Archiv):

```
$ sh Configure -des -Dusedevel
$ make
$ make test
```

Die `-d -e -s`-Optionen bedeuten im Prinzip: „Frage mich nicht irgendwelche Fragen und verwende vernünftige Standardeinstellungen für alles!" Die `-Dusedevel`-Option bedeutet einfach nur, dass `Configure` sich nicht darüber beschweren soll, dass du eine Entwicklerversion von Perl baust, wenn du aus einem *git clone* heraus arbeitest. Es ist im Grunde die „Ja, ich will es wirklich"-Option, die verhindern soll, dass Leute eine nicht-veröffentlichte Version von Perl in Produktivsystemen ausrollen. Um Dein Perl auf einer Multi-Core-Maschine schneller zu bauen und zu testen, kannst du einige `-j`-Magie verwenden:

```
$ sh Configure -des -Dusedevel
$ TEST_JOBS=5 make -j5 test
```

Auf diese Weise wird mit fünf parallelen Jobs kompiliert und getestet. Wenn du das neue Perl an einen bestimmten Ort installieren willst, dann füge noch `-Dprefix=/home/you/mydebugperl` hinzu und rufe `make install` auf. Auf dem Weg zu einem Perl mit besseren Debugging-Möglichkeiten, sollen für den Anfang Debugging-Symbole in der Ausgabe eingebunden und möglicherweise die C-Compiler-Optimierungen ausgeschaltet werden. Dann füge `-Doptimize="-g -O0"` zum `Configure`-Aufruf hinzu. Das ist praktisch zum Aufspüren von Problemen im Perl-Code, wenn du auf die *valgrind*-Ausgabe schaust. Als nächstes wird ein Perl gebaut, bei dem seine eigenen Debugging-Funktionalitäten aktiviert sind: Füge `-DDEBUGGING` hinzu. Fassen wir alles bisherige zusammen, bekommen wir:

```
$ sh Configure -des
 -Dprefix=/home/you/mydebugperl -Dusedevel \
  -Doptimize="-g -O0" -DDEBUGGING
```

Alles, was du bisher erreicht hast, ist natürlich eine Kopie von Perl, die massiv langsamer ist als dein produktives Perl (wahrscheinlich eine ganze Größenordnung langsamer) und die dir noch nicht wirklich dabei helfen wird, dein Referenzzähler-Problem zu debuggen. So ein Perl zu haben, ist das typische Sprungbrett für das Debuggen des Perl-Kerns und bis jetzt sind die Schritte an anderer Stelle sehr gut dokumentiert.

Das *perlhacktips*-Dokument erklärt eine Reihe von komplizierteren Optionen für das Speicherdebugging. Der Ab-

schnitt *PERL_DESTRUCT_LEVEL* ist von besonderem Interese. Es stellt sicht heraus, dass Perl sich standardmäßig nicht um das Säubern der Speicher-*slabs* kümmert, wenn es fertig ist. Es lässt das im Allgemeinen das Betriebssystem erledigen (ich glaube, weil es weniger Overhead hat). Das Setzen der Umgebungsvariablen `PERL_DESTRUCT_LEVEL` während der Programmausführung lässt Perl pedantischer werden. Es ist wichtig, solche Sachen für Tools wie *valgrind* von Anfang an ersichtlich zu machen:

```
$ PERL_DESTRUCT_LEVEL=
 2 perl your_buggy_program.pl
```

Wenn du das Gegenteil des „Attempt to free..."-Problem hattest, also SVs mit zu hohem Referenzzähler, dann bekommst du mit diesem Setup Benachrichtigungen über leckende Skalare. Der nächste Schritt zum Grund des Problems schließt die `Configure`-Optionen `-DDEBUG_LEAKING_SCALARS`, `-DDEBUG_LEAKING_SCALARS_FORK_DUMP` und `-DDEBUG_LEAKING_SCALARS_ABORT` ein. Diese sind größtenteils in *perlhacktips* dokumentiert.

Um das vermutete Referenzzähler-Problem einfacher aufzufinden, können wir eine Option angeben, die für das `Purify`-Tool gedacht ist: `-Accflags=-DPURIFY` (das heißt: Füge *-DPURIFY* zu den C-Compiler-Optionen hinzu. Mit dieser C-Definition wird Perl keine *slabs* für das Zuweisen von SVs verwenden, was deine Chancen erhöhen sollte, seltsames Verhalten aufzuspüren. Zusätzlich können wir Perl anweisen, freigegebene Speicherbereiche mit einem bekannten Muster (`0xEF`) zu überschreiben. Das verhindert, dass Fehler durch das Wiederverwenden von Speicher, der vorher für SVs benutzt wurde, verschleiert werden. Ähnlich wie die Purify-Option ist das auch eine C-Compiler-Definition. So bekommen wir: `-Accflags="-DPURIFY -DPERL_POISON"`.

Nur um alle Tools zusammenzubringen, zeige ich hier, wie ich schließlich mein Perl gebaut habe. Die Einstellungen `-Dcc` und `-Dld` erlauben es mir, *ccache* mit *gcc* für schnelleres wiederholtes Kompilieren zu verwenden:

```
$ sh Configure -Doptimize="-g -Wall
     -Wextra -O2" -DDEBUGGING -Dusedevel  \
 -Dprefix=/home/you/mydebugperl
     -Dcc=ccache\\ gcc\\ -g -Dld=gcc \
 -Uusethreads -de -DPERL_TRACK_MEMPOOL
     -DDEBUG_LEAKING_SCALARS_FORK_DUMP \
 -DDEBUG_LEAKING_SCALARS -Accflags=
    "-DPURIFY -DPERL_POISON" \
 -DDEBUG_LEAKING_SCALARS_ABORT
```

Mein unreferenzierter Skalar

Ein Perl, das mit allen oben genannten Debugging-Tools ausgestattet ist, hat meine Debugging-Bemühungen in vielen Fällen viel einfacher gemacht. In der Regel ist ein Teil der oben gezeigten Möglichkeiten ausreichend. Leider hat es mir in diesem Fall nur das sorgfältige Durchgehen meines Codes und Dumpen der Adressen von vielen SVs, um den einen zu finden, der zu früh freigegeben wurde, erlaubt, den einen einzelnen Befehl zu finden, der mir den Tag verdorben hat:

```
SvREFCNT_dec(*fetched_sv);
```

Der Code hat einen Hash-Zugriff verwendet, der das Element erzeugt, wenn es beim Zugriff nicht existiert (`HV_FETCH_LVALUE|HV_FETCH_JUST_SV`-Modus von *hv_common*), aber fälschlicherweise davon ausgegangen ist, dass ein Inkrementieren des Referenzzählers Teil der Operation ist. Unnötig zu sagen, dass ich den Fehler beseitigt habe und dann einen schwierigen Siegestanz aufgeführt habe.

Der eigentliche Fehler wurde nur durch viele, viele Millionen Tests zu Tage gebracht, die gegen unsere Perl/XS-Implementierung der Bibliothek für Serialisierung und Deserialisierug zum Schutz gegen Attacken laufen. Aber das ist ein anderes Thema.